



Applications of Artificial Neural Networks in Structural Engineering with Emphasis on Continuum Models

by *Rakesh K. Kapania** and *Youhua Liu***

Department of Aerospace and Ocean Engineering
Virginia Polytechnic Institute and State University

Blacksburg, VA 24061-0203

June, 1998

*: Professor

** : Graduate Research Assistant

Abstract

The use of continuum models for the analysis of discrete built-up complex aerospace structures is an attractive idea especially at the conceptual and preliminary design stages. But the diversity of available continuum models and hard-to-use qualities of these models have prevented them from finding wide applications. In this regard, Artificial Neural Networks (ANN or NN) may have a great potential as these networks are universal approximators that can realize any continuous mapping, and can provide general mechanisms for building models from data whose input-output relationship can be highly nonlinear.

The ultimate aim of the present work is to be able to build high fidelity continuum models for complex aerospace structures using the ANN. As a first step, the concepts and features of ANN are familiarized through the MATLAB NN Toolbox by simulating some representative mapping examples, including some problems in structural engineering. Then some further aspects and lessons learned about the NN training are discussed, including the performances of Feed-Forward and Radial Basis Function NN when dealing with noise-polluted data and the technique of cross-validation. Finally, as an example of using NN in continuum models, a lattice structure with repeating cells is represented by a continuum beam whose properties are provided by neural networks.

1. Introduction

It is estimated that about 90% of the cost of an aerospace product is committed during the first 10% of the design cycle. As a result, the aerospace industry is increasingly coming to the conclusion that physics-based high fidelity models (Finite Element Analysis for structures, Computational Fluid Dynamics for aerodynamic loads etc.) need to be used earlier at the conceptual design stage, not only at a subsequent preliminary design stage. But an obstacle to using the high fidelity models at the conceptual level is the high CPU time that are typically needed for these models, despite the enormous progress that has been made in both the computer hardware and software.

During the late seventies and early eighties, there was a significant interest in obtaining continuum models to represent discreet built-up complex lattice, wing, and laminated wing structures. These models use very few parameters to express the original structure geometry and layout. The initial model generation and set-up is fast as compared to a full finite element model. Assembly of stiffness and mass matrices and solution times for static deformation and stresses or natural modes are significantly less than those needed in a finite element analysis. All these make continuum models very attractive for preliminary design and optimization studies.

Despite its great potential, the continuum approach has gained a limited popularity in the aerospace designers community. This is, perhaps, due to the fact that all the developments have been made by keeping specific examples (eg. periodic lattices or specific wings) in mind. Also, with some exceptions, most of these approaches were rather complex. The key obstacle though appears to be the fact that if the designer makes a change in the actual built-up structures, the continuum model has to be determined from

scratch.

The complex nature of the various methods and the large number of problems encountered in determining the equivalent models are not surprising given the fact that determining these models for a given complex structure (a large space structure or a wing) belongs to a class of problems called inverse problems. These problems are inherently ill-posed and it is fruitless to attempt to determine unique continuum models. The present work deals with investigating the possibility that a more rational and efficient approach of determining the continuum models can be achieved by using artificial neural networks.

The working mechanism in brains of biological creatures has long been an area of intense study. It was found around the first decade of this century that *neurons* (nerve cells) are the structural constituents of the brain. The neurons interact with each other through *synapses*, and are connected by *axons* (transmitting lines) and *dendrites* (receiving branches). It is estimated that there are on the order of 10 billion neurons in the human cortex, and about 60 trillion synapses (Ref. 1). Although neurons are 5~6 orders of magnitude slower than silicon logic gates, the organization of them is such that the brain has the capability of performing certain tasks (for example, pattern recognition, and motor control etc.) much faster than the fastest digital computer nowadays. Besides, the energetic efficiency of the brain is about 10 orders of magnitude lower than the best computer today. So it can be said, in the sense that a computer is an information-processing system, the brain is a highly complex, nonlinear, and efficient parallel computer.

Artificial Neural Networks (ANN), or simply Neural Networks (NN) are computational systems inspired by the biological brain in their structure, data processing and restoring method, and learning ability. More specifically, a neural network is defined as a massively parallel distributed processor that has a natural propensity for storing ex-

periential knowledge and making it available for future use by resembling the brain in two aspects: (a) Knowledge is acquired by the network through a learning process; (b) Inter-neuron connection strengths known as synaptic weights (or simply weights) are used to store the knowledge (Ref. 1).

With a history traced to the early 1940s, and two periods of major increases in research activities in the early 1960s and after the mid-1980s, ANNs have now evolved to be a mature branch in the computational science and engineering with a large number of publications, a lot of quite different methods and algorithms and many commercial software and some hardware. They have found numerous applications in science and engineering, from biological and medical sciences, to information technologies such as artificial intelligence, pattern recognition, signal processing and control, and to engineering areas as civil and structural engineering.

In this report a brief description is given to the most extensively used neural network in civil and structural engineering, Multi-Layer Feed-Forward NN, and another kind of NN, Radial Basis Function NN, which is very efficient in some cases. Some conceptual features of NN are listed. Some examples of the application of neural network are given, among which several are published real problems in civil and structural engineering. Based on our experience of using the MATLAB NN Toolbox, some important and very practical issues on the application of NN will be brought out.

Then a section of the report is dedicated to the development of algorithms carrying out the very useful concept of cross-validation. Through the results for several examples obtained from the algorithms, some observations on issues such as over-training and network complexity are given.

In the last section an example of using NN in continuum models is given. A lattice structure with repeating cells is represented by a continuum beam whose properties, as functions of the repeating cell particulars, are provided by neural networks.

2. Examples of NN

As simplified models of the biological brain, ANNs have lots of variations due to specific requirements of their tasks by adopting different degree of network complexity, type of inter-connection, choice of transfer function, and even differences in training method.

According to the types of network, there are Single Neuron network (1-input 1-output, no hidden layer), Single-Layer NN or Perceptron (no hidden layer), and Multi-Layer NN (1 or more hidden layers). According to the types of inter-connection, there are Feed-Forward network (values can only be sent from neurons of a layer to the next layer), Feed-Backward network (values can only be sent in the different direction, i.e. from the present layer to the previous layer), and Recurrent network (values can be sent in both directions).

In the following a brief description is given to two kinds of extensively used neural networks and some of the pertinent concepts.

2.1 Feed-Forward Multi-layer Neural Network

As shown in Fig. 1(a), in the j -th layer, the i -th neural has inputs from the $(j - 1)$ -th layer of value $x_k^{j-1} (k = 1, \dots, n_{j-1})$, and has the following output

$$x_i^j = f(\tau_i^j)$$

where

$$r_i^j = \sum_{k=1}^{n_{j-1}} w_{ki}^{j-1} x_k^{j-1} - b_i^j$$

in which w_{ki}^{j-1} is the weight between node k of the $(j-1)$ -th layer and node i of the j -th layer, and b_i^j is the bias (also called threshold). The above relation can also be written as

$$r_i^j = \sum_{k=0}^{n_{j-1}} w_{ki}^{j-1} x_k^{j-1}$$

where $x_0^{j-1} = -1$ and $w_{0i}^{j-1} = b_i^j$.

The *transfer function* (also called *activation function* or *threshold function*) is usually specified as the following *Sigmoid function*

$$f(r) = \frac{1}{1 + e^{-r}}.$$

Other choices of the transfer function can be the *hyperbolic tangent function*

$$f(r) = \frac{1 - e^{-r}}{1 + e^{-r}},$$

the *piecewise-linear function*

$$f(r) = \begin{cases} 1, & r \geq 0.5; \\ r + 0.5, & -0.5 \leq r \leq 0.5; \\ 0, & r \leq -0.5. \end{cases}$$

and, sometimes, the '*pure*' *linear function*

$$f(r) = r.$$

These transfer functions are displayed in Fig. 1(b).

2.2 Radial Basis Function Neural Network

Radial Basis Function (RBF) NN usually have one input layer, one hidden layer and one output layer, as shown in Fig. 2.

For the RBF network, Fig. 2, we have the relations between the input x_i^1 (here $i=1, \dots, n_1$) and the output x_k^2 (here $k=1, \dots, n_3$) as follows.

$$x_k^2 = \sum_{j=1}^{n_2} w_{k,j}^2 r_j + b_k^2$$

$$r_j = \sum_{i=1}^{n_1} G(x_i^1, w_j^1, b_{i,j}^1)$$

where w^2, b^2 are the weights and bias respectively, and the *Gaussian function* is used as the transfer function:

$$G(x_i^1, w_j^1, b_{i,j}^1) = \exp(-[b_{i,j}^1]^{-2} [x_i^1 - w_j^1]^2)$$

where w^1 is the center vector of the input data, and b^1 is the variance vector.

3. Features of ANN

Some important features of NN are briefed as follows.

- Many NN methods are universal approximators, in the sense that, given a dimension (number of hidden layers and neurons of each layer) large enough, any continuous mapping can be realized. Fortunately, the two NNs we are most interested in, the multi-layer feed-forward NN and the radial basis NN, are examples of such universal approximators (Ref. 4,5).

- Steps of utilizing NN: **specification of the structure (topology) → training (learning) → simulation (recalling).**

- (1) Choosing structural and initial parameters (number of layers, number of neurons of each layer, and initial values of weights and thresholds, and the kind of transfer function) is usually from experiences of the user and sometimes can be provided by the algorithms.
- (2) The training process uses given input and output data sets to determine the optimal combination of weights and thresholds. It is the major and the most time-consuming part of NN modeling, and there are lots of methods regarding different types of NN.
- (3) Simulation means using the trained NN to predict output according to new inputs (This corresponds to the 'recall' function of the brain).

- The input and output relationship of NN is highly nonlinear. This is mainly introduced by the nonlinear transfer function. Some networks, e.g. the so-called "abductive" networks, use double even triple powers besides linear terms in their layer to layer input-output relations (Ref. 6).

- A NN is parallel in nature, so it can make computation fast. Neural networks are ideal for implementation in parallel computers. Though NN is usually simulated in ordinary computers in a sequential manner.

- A NN provides general mechanisms for building models from data, or give a general means to set up input-output mapping. The input and output can be continuous (numerical), or not continuous (binary, or of patterns).

- Training a NN is an optimization process based on minimizing some kind of difference between the observed data and the predicted while varying the weights and thresholds. For numerical modeling, which is of our major concern for the present study, there is a great similarity between NN training and some kind of least-square fitting or interpolation.

- Where and when to use NN depend on the situation, and NN is not a panacea. The following comment on NN application on structural engineering seemingly can be generalized in other areas:

"The real usefulness of neural networks in structural engineering is not in reproducing existing algorithmic approaches for predicting structural responses, as a computationally efficient alternative, but in providing concise relationships that capture previous design and analysis experiences that are useful for making design decisions" (Ref. 9).

4. Algorithms in the MATLAB Neural Network Toolbox

When using MATLAB NN Toolbox, one should first choose the number of input and output variables. This is accomplished by specifying the two matrices p and t ; where p is a $m \times n$ matrix; m is the number of input variables, and n the number of sets of training data; and t is a $l \times n$ matrix; l is the number of output variables. Then the number of network layers, and numbers of neurons of each layer should be specified.

MATLAB gives algorithms for specifying initial values of weights and thresholds in order that training can be started. For feed-forward NN, function *initff* is given for this purpose. The following is an example of using the algorithm

$$[w1, b1, w2, b2, w3, b3] = \text{initff}(p, n1, 'logsig', n2, 'logsig', t, 'logsig');$$

where $w1$, $w2$, $w3$ are initial values for the weight matrices of the 1st (hidden), 2nd (hidden) and 3rd (output) layer respectively, $b1$, $b2$, $b3$ are the bias (threshold) vectors, $n1$ and $n2$ the number of neurons in the 1st and 2nd hidden layer respectively, and 'logsig'

means that the Sigmoid transfer function is used.

The present version of MATLAB NN Toolbox can support only 2 hidden layers, but the number of neurons is only limited by the available memory of the computer system being used. For the transfer function, one can also use other choices, such as '*tansig*' (hyperbolic tangent sigmoid), '*radbas*' (radial basis) and '*purelin*' (linear) etc.

Experiences of using *initff* indicated that it seems to be a random process since it is found that the result of the execution of this algorithm each time is different. And other conditions kept the same, two execution of this function usually give quite different converging histories of training by the training algorithm (see Fig. 3 (a) and (b)). We shall discuss this later in 6.3.

Shown in the following is the MATLAB algorithm for training feed-forward network with back-propagation:

$$[w1,b1,w2,b2,w3,b3,ep,tr]=trainbp(w1,b1,'logsig',w2,b2,'logsig', w3,b3,'logsig',p,t,tp);$$

where most of the parameters which the user should take care of have been mentioned in the above paragraphs. The only set of parameters that the user sometimes need to specify is the 1×4 vector *tp*, where the first element indicates the number of iterations between updating displays, the second the maximum number of iterations of training after which the algorithm terminates the training process, the third the converging criterion (sum-squared error goal), and the last being the learning rate. The default value of *tp* is [25, 100, 0.02, 0.01].

Other algorithms for training: *trainbpa* (train feed-forward NN with back-propagation and adaptive learning), *solverb* (design and train radial basis network), and *trainlm* (train

feed-forward NN with Levenberg-Marquardt) etc.

trainbpa and *trainlm* have very similar formats for using as that of *trainbp*. The radial basis network designing and training algorithm has the following format

$$[w1, b1, w2, b2, nr, err] = solverb(p, t, tp).$$

where the algorithm chooses centers for the *Gaussians* and increases the neuron number of the hidden layer automatically if the training cannot converge to the given error goal. So it is also a designing algorithm.

After the NN is trained, one can predict output from input by using simulation algorithms in terms of the obtained parameters *w1*, *b1*, *w2*, *b2*, etc. For feed-forward network one use

$$y = \text{simuff}(x, w1, b1, 'logsig', w2, b2, 'logsig')$$

where *x* is the input matrix, and *y* the predicted output matrix. Similarly, after a radial basis network has been trained one uses

$$y = \text{simurb}(x, w1, b1, w2, b2)$$

to predict the output.

Once a NN is trained, we can use the formulations in 2.1 or 2.2 together with the obtained parameters (weights etc.) to setup the network to do prediction anywhere and not necessarily within the MATLAB environment.

5. Examples of Application Using MATLAB Neural Network Toolbox

5.1 A Single Input Single Output Function

The training data set are

$$p=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0];$$

$$t=[0.01, 0.04, 0.09, 0.16, 0.25, 0.36, 0.49, 0.36, 0.25, 0.16];$$

A 1-9-1 (a input layer with 1 input, a hidden layer with 9 neurons, and an output layer with 1 output) feed-forward NN was trained with *trainbp*. The comparison between the desired (with symbol '+') and the predicted (smooth line) values and the training history are shown in Figs. 4 (a) and (b) respectively. Another 1-9-6-1 (an input layer with 1 input, two hidden layer with 9 and 6 neurons respectively, and a output layer with 1 output) feed-forward NN was also trained, and the comparison between the desired values and the predicted values and the training history are shown in Figs. 5 (a) and (b), respectively.

By adjusting the learning rate according to the circumstances, *adaptive learning* usually can give better results than with a constant learning rate specified before the training begins, as displayed in Fig. 6 (a) and (b). In Fig. 6 (b), the dotted curve is the variation of learning rate, and the continuous curve is the variation of error.

5.2 Training of Multiple Variable Mappings

A multi-variable mapping is much more complicated than a single input single

output function. If we say for the single input single output function in Sect. 5.1, 10 sets of data can give a somewhat complete description of the relation between the input and the output, for a mapping with k input variables and one output variable, we will need 10^k sets of data to obtain the same degree of completeness of description. That means the efforts of training a neural network to simulate a multiple variable mapping increases exponentially with the number of input variables.

For many practical cases, one must obtain a multiple variable mapping from a quite limited data set. The worse is, the data are usually randomly distributed in the input domain, rather than distributed uniformly as in the cases of experiments designed by the DOF (degree of freedom) method. It will be of interest to know the behaviors of NNs for such kind of data.

The mapping shown in Fig. 7 is

$$z = \frac{2}{5}(1 + \sin(4xy^2)), D = \{0 \leq x, y \leq 1\}.$$

Firstly we used an 11×11 training data uniformly distributed in the domain D to train a 2-10-1 feed forward NN, then the NN was used to simulate the mapping on a 51×51 testing mesh uniformly distributed in the domain D . The simulated mapping and the relative errors, compared to the exact mapping shown in Fig. 7, are given in Figs. 8(a) and 8(b) respectively. It can be noted that around a corner of domain D , i.e. $(x=1, y=1)$, the relative errors have larger values since the abstract values of z have a minimum at the corner.

Secondly we trained a 2-10-1 NN from a training data randomly distributed in the domain D with the number of data sets $n_0 = 11 \times 11 = 121$. The NN was also used to simulate the mapping on the 51×51 testing mesh uniformly distributed in the domain D ,

and the results are shown in Fig. 9. Comparing Fig. 9(b) with Fig. 8(b), we can see that a NN trained with randomly distributed data has an accuracy at least one order of magnitude worse than that with uniformly distributed training data.

Then more NNs were trained from training data randomly distributed in the domain D with the number of data sets $n = \eta n_0$ and used to simulate the mapping on the 51×51 testing mesh uniformly distributed in the domain D . One example of these results, $n=24$, is shown in Fig. 10. The error (the maximum of the 51×51 relative testing errors) with randomly distributed training data as a function of η , compared with the results obtained using uniformly distributed training data, is shown in Fig. 11. From these results the following conclusions can be drawn:

(a) With uniformly distributed training data, the accuracy of the NN is much higher than that with randomly distributed training data;

(b) For cases with randomly distributed training data, when the number of training data sets decreases, the averaged relative error increases in an inversely proportional way, and the error deviation increases either.

5.3 A Neural Network to Represent Experimental Data

This is an application of NN to represent experimental data fitting to model a structural problem, namely, the behavior of semi-rigid steel structure connections related to the physical connection conditions of the adjacent parts (contact and friction), and local plastification effects etc. While these highly nonlinear effects prevent a practical rational method from being available, neural network modeling from some experimental data base seems to be a good solution to the related structural design and analysis problems.

Only the case of single-angle beam-to-column connections bolted to both beam and column (Fig. 12) would be reproduced. There are 6 sets of experimental data available, 5 were used for training, and the remaining 1 set for testing. The input data are shown in Table 1.

Following Ref. 7, a 3-50-50-22 feed-forward neural network is used. The network has 3 input variables, given in Table 1, two hidden layers of 50 neurons each, and 22 output variables (the first 21 are moments normalized by the largest at different angles of rotation, as specified in Fig. 5 of Ref. 7, and the last variable is a value reflecting the largest moment of each set of data, an inverse of the value of the largest moment multiplying by a constant making sure that result being less than 1).

Results with different choices of training sets or different initialization schemes are shown in Fig. 13.(a), (b), and(c). Both in (a) and (c) Experiment No.1, 2, 3, 5, 6 are used as training sets, and No. 4 is used as the testing set. While in (b) Experiment No.1, 2, 3, 4, 6 are used as training sets, and No. 5 is used as the testing set. The difference between (a) and (c) is that in (a) '*logsig*' is chosen as the transfer function in using the algorithm *initff* to provide initial weights and bias for the training process (the same as in (b)), while in (c) '*tansig*' is used. In all the three cases '*logsig*' is used as the transfer function in the training process. The error goal for the training is 0.5E-5, and the number of iterations for convergence were 12377, 6991 and 17118 respectively, in the three cases.

As shown in Fig. 13, the NN with adequate hidden layers and neurons is flexible enough to give almost perfect reproduction of the training data. This is anticipated if the training process is converged to the specified small error goal. But what is of a great interest is the comparison between the actual and the predicted values for the testing data sets, the ones which were not used for the training process. We can see, for

the steel connection example, the NN modeling is not reliable, the predicted result can either overestimate [as in (a)], underestimate [as in (b)], or give good approximation of the desired value [as in (c)]. This is mainly due to the fact that different combinations of transfer functions are used. As will be seen in the following sections (6.3 and 5.5), when the transfer function '*tansig*', instead of '*logsig*', is used in the algorithm *initff*, together with '*logsig*' used in the training algorithm *trainbp*, (that is, *Formulation II* instead of *Formulation I* in 6.3), the network is more robust to give smooth approximations, and more importantly, a much better testing accuracy is obtained. Another reason might be that the input data lack enough variances to indicate their influence on the relationship and cover the range of practical possibilities (especially for the angle thickness).

5.4 Neural Networks Applied to More Structural Problems

More applications of neural network in structural engineering can be found in Ref. 8. All the problems treated in Ref. 8 had been reproduced in Ref. 9 with a conclusion that representational change of a problem based on dimensional analysis and domain knowledge can improve the performance of the networks. We shall reproduce the results of the first two problems with an aim to compare the performances of MATLAB NN Toolbox and those of the software used in Ref. 8 and Ref. 9, and to increase our experiences of dealing with real structural problems.

5.4.1 A Pattern Recognition Problem

The first problem is related to pattern recognition. The neural network will be trained using the bending moment patterns in a simply supported beam subjected to a concentrated load at several different locations. The input variables are the bending

moments at 10 sections along the span, and the location of the load represents the output value. As mentioned in Ref. 9, the training data sets in Ref. 8 have some mistakes in them. The data sets shown in Table 2 are the corrected ones.

We use both multi-layer feed-forward network with backward propagation training and the radial basis function network to attack the problem. A 10-3-1 feed-forward NN is used, and the results are shown in Fig. 14. With only one hidden layer and 3 neurons in the hidden layer, we have produced results that are comparable to those in Refs. 8 and 9 achieved using 2 hidden layers, each with 10 neurons. This seems to be because our neuron number has already been of the order of the number of training data sets (the latter is 4).

The radial basis NN performed better for this problem. With only 3 iterations (which means that the MATLAB algorithm solverb has changed the network structure only 3 times, each time increasing a neuron in the hidden layer, until 3 sets of the training data have been taken as the center vector), it gives results (see Fig. 15) which are comparable with those obtained from the feed-forward network after thousands of iterations. This is consistent with the finding that radial basis networks behave well for modeling systems with low dimension or isolated "bumps" (Ref. 3) (the pattern of the beam moment distribution is bump-shaped).

5.4.2 An Ultimate Moment Capacity of a Beam

The second problem is about the design of a simple concrete beam, that is, selecting the depth of a singly reinforced rectangular concrete beam to provide the required ultimate moment capacity.

5.4.2.1 An Multi-Input Single Output Formulation

In Ref. 8, the problem was formulated into a mapping with 5 input variables and 1 output variable. A feed- forward network with two hidden layers of six neurons each was used. A set of 21 patterns were randomly chosen as training data. And for the testing, a set of 10 additional patterns were used (see Table 3 in Ref. 8). We used the same settings for training the network and the same training and testing data. The results are displayed in Fig. 16(a) as compared with those of Ref. 8 in Fig 16(b).

5.4.2.2 A Single Input Single Output Formulation

It has been pointed out in Ref. 9 that from the non-dimension analysis of the mathematical model, only two non-dimensional variables are required for the representation. Hence the data used in Ref. 8 were reorganized in Ref. 9 and a network with one input and one output neuron, and one hidden layer having 3 neurons was used. This is a simpler relationship, and it was found in Ref. 9 that by using only five of the patterns as the training data, the network can produce very good predictions. We use the same training and testing data (Table 3 in Ref. 9), and the settings of the network are almost the same (except more neurons and one more hidden layer are tried). One of the results is shown in Fig. 17 (a), which is comparable to those obtained in Ref. 9 as displayed in Fig. 17 (b).

There has been a significant experience in using MATLAB NN Toolbox to deal with this seemingly simple mapping problem, as we shall discuss in the following.

5.5 An Example of NN Modeling to Data with Noises

The example problem used in Ref. 11 will be solved here by both feed-forward NN and radial basis function (RBF) NN to show their respective abilities of modeling data with noises.

A set of 51 points sampled uniformly in the x-direction but with randomly chosen noises in the y-direction can be generated by the MATLAB code as follows.

```
p=51;
x=0:.02:1;
sigma=0.15;
y=.5+.4*sin(10*x)+sigma*(.5-rand(1,p));
plot(x,y,'c+')
```

The true function, $y(x)=.5+.4\sin(10x)$, which will be generated with a more dense set of points and plotted by dashed line, is sampled without noise.

```
pt=500;
xt=linspace(0,1,pt);
yt=.5+.4*sin(10*x)+sigma*(.5-rand(1,p));
hold on
plot(xt,yt,'b-')
```

The comparison of the training points with noises specified above, feed-forward NN simulation with two hidden layers (1-20-20-1), and the true function is shown in Fig. 18(a). A similar comparison with a RBF NN simulation is shown in Fig. 18(b). We can see that in both cases that the neural networks give reasonable and smooth simulation of the true function even though the training data has noise.

Another set of training points is generated with noises in both x- and y- direction. This gives an even worse situation for the neural network to deal with.

```
p=51;
x=0:.02:1;
sigma1=0.01;
sigma2=0.15;
x=x+sigma1*(.5-rand(1,p));
y=.5+.4*sin(10*x)+sigma2*(.5-rand(1,p));
plot(x,y,'c+')
```

Comparison of the results in this case is shown in Fig. 19(a) through (d), where (a) is about the results by a (1-20-20-1) Feed-Forward NN, (b) is about a RBF NN simulation, (c) is a summation of the results in (a) and (b), and (d) is about the results by a (1-40-1) FF NN trained with adaptive learning. Very good results are also obtained by all neural network modeling. As in the first case, the RBF neural network gives the best results.

Shown in Fig. 20 is the result from the feed-forward NN with transfer function combination *Formulation I* (see 6.3 in the following). In this case, it is difficult for the training to converge, and back-propagation with adaptive learning (that means the learning rate is adjusted in each epoch) has to be adopted. Moreover, the simulated result curve is quite rough for both 1 and 2 hidden layer cases. We shall discuss this further in 6.3.

6. On Some Issues of Neural Network Modeling

6.1 About the Accuracy and Error Goal

In a sense NN modeling is something similar to fitting a curve or interpolation. It seems almost impossible that in the presence of a limited information set (the training data) a NN can simulate the performance of a system under all possible input scenarios. Quite often, both the training and the input data sets include all sorts of noises.

Generally, it is expected that the accuracy of a NN could be better in the case of "interpolation" than in the case of "extrapolation" (for instance, see Figs. 14 and 15).

The error goal for the training of a NN has a direct impact on the training time and some influence on the prediction accuracy. It seems that since usually the data (whether for training or for testing) is polluted by noise, a very small error goal would not be a good strategy for the training process. Besides a long training time or even failure to converge, that sometimes may mean worse accuracy for the testing data outside the training set.

These have been clearly shown in Fig. 21. In Fig. 21(a) we can see that as a result of very small error goal, the accuracy of testing is very poor despite a perfect training accuracy, and in Fig. 21(b) and (c) through the choice of a moderate error goal, both the testing and training accuracy are of the same order. For the case in Fig. 21(a) the training time was about 5 times longer.

6.2 Normalization of the Training Data

When the Sigmoid or other transfer functions which squash the output in the range $[-1,1]$ are used, preparation of data is very important before modeling by NN and training. For feed-forward neural networks, the input and output data must be normalized, that is to say, they must be scaled to values less than 1. This is because the Sigmoid function cannot give values larger than 1. Usually the largest value should be kept to a value that

is close and less than 1, say 0.98. If the original values of the input and output are much less than 1, it should also be normalized and scaled to a value close to 1.

We can see this effect clearly in Fig. 22 (a) and (b). In Fig. 22(a) the ultimate moment capacity problem (5.4.2.2) is again solved by multiplying input p by 4 and output t by 3 so that both the largest values are close to 1. Compared with Fig. 17(a) (where the number of iterations (epoch) is in thousands) the performance of the NN has improved, especially over the range of x from 0.16 to 0.19. In Fig. 22(b) the result of a single input single output problem using the formulation

$$[w1, b1, w2, b2, ep, tr] = \text{trainlm}(w1, b1, 'logsig', w2, b2, 'logsig', p, t, tp);$$

is shown, where we can see that the prediction cannot approach the point that surpasses 1.

The situation can be overcome when the 'pure' linear transfer function is used. This is illustrated in Fig. 22(c), where the result for the same problem in Fig. 22(b) using the formulation

$$[w1, b1, w2, b2, ep, tr] = \text{trainlm}(w1, b1, 'logsig', w2, b2, 'purelin', p, t, tp);$$

is shown.

For the radial basis NN, although there is no such restriction, still it is better to scale the input to and output from such a network around (0,1) for an improved performance.

6.3 Initialization and Specification of Transfer Function

There are some tricks in using the initialization algorithms. Firstly, as already mentioned, the initialization algorithm *initff* is a random process, so it is possible that

training histories of a given problem may be quite different, with one showing a quick convergence and the other needing thousands of extra iterations.

Secondly, the specification of the transfer function involved in *initff* and *trainbp* has very large influences on the training history and the performance of the network. It has been found that there are only two combinations of *initff* and *trainbp* that are feasible in practice, as shown below:

Formulation I:

$$[w1,b1,w2,b2,w3,b3]=initff(p,n1,'logsig',n2,'logsig',t,'logsig');$$

$$[w1,b1,w2,b2,w3,b3,ep,tr]=trainbp(w1,b1,'logsig',w2,b2,'logsig', w3,b3,'logsig',p,t,tp);$$

Formulation II:

$$[w1,b1,w2,b2,w3,b3]=initff(p,n1,'tansig',n2,'tansig',t,'tansig');$$

$$[w1,b1,w2,b2,w3,b3,ep,tr]=trainbp(w1,b1,'logsig',w2,b2,'logsig', w3,b3,'logsig',p,t,tp);$$

Formulation I usually makes the training process oscillate (the error-epoch line quite rough) but approach the error goal more directly while the error-epoch line of *Formulation II* is usually quite smooth but at the later part of training the error decreases very slowly. See Figs. 4(b), 5(b) (*Formulation II*), and Fig. 3(a), 3(b) (*Formulation I*)

As to the performance of a NN, although *Formulation I* can predict the output in the training data set quite well, it sometimes gives very poor testing results outside the training data. See Fig. 23 and Fig. 13 (a) and (b). Further, from Fig. 18 to 20, we can see while *Formulation II* gives quite robust and smooth performance *Formulation I*, in contrast, gives unstable and rough predictions when the training data includes noise.

So we prefer *Formulation II*.

6.4 NN Structural Problem

The number of the hidden layers and the number of neurons in each layer determine the freedom or dimension of a NN. Generally speaking, more hidden layers and more neurons in the layers mean a larger flexibility of the NN, and accordingly smaller error goal can be specified.

The relation between the training time and the numbers of hidden layers and neurons is a complicated one. A more complex NN (with more hidden layers and neurons) certainly needs more execution time in each training iteration. But a network with more layers or neurons may need less iterations for training to reach the desired error goal.

From Fig. 24 (a) through (h), we can say the adequate degree of freedom of a NN (total number of neurons) is about the same order of the number of the training data sets. More hidden layers and neurons not only increase the training effort, but sometimes also give poor performance for the network.

7. Cross-Validation Algorithms and Some Observations

Cross-validation is a standard tool in statistics (Ref. 1). This technique can be used to optimize neural network structural parameters, i.e. numbers of hidden layers and neurons of each hidden layer etc. It can also be utilized to determine the proper training epoch in order to avoid over-training, the case that usually occurs when the number of training iteration is too high, with the symptom that errors for the training data approach

zero while those for the testing data increase excessively.

When a network is being trained using the cross-validation technique, the training data is divided into two subsets, one subset being used for training of the network, the other being used for validation (i.e. testing). The validation subset is typically 10 to 20 percent of the training set.

For the case of single output, the error can be defined as the following *combined error*

$$E_{com} = \left(\frac{1}{n_r + n_s} \left(\sum_{i=1}^{n_r} (t_{ri} - s_{ri})^2 + \sum_{i=1}^{n_s} (t_{si} - s_{si})^2 \right) \right)^{1/2}$$

where n_r and n_s are numbers of data sets in the training and testing subset respectively, t_r and t_s are the target value for the training and testing subset, and s_r and s_s are the values predicted by the network for the training and testing subset. Other two error measures are:

$$E_{tr} = \left(\frac{1}{n_r} \left(\sum_{i=1}^{n_r} (t_{ri} - s_{ri})^2 \right) \right)^{1/2}$$

and

$$E_{ts} = \left(\frac{1}{n_s} \left(\sum_{i=1}^{n_s} (t_{si} - s_{si})^2 \right) \right)^{1/2}$$

are also of interest and so is the error ratio E_{ts}/E_{tr} .

MATLAB algorithms *cvtrain*, *cvtrain1* are available to train Feed-forward NN using cross-validation. They are invoked in the following format

$$[w1,b1,w2,b2,w3,b3]=cvtrain(n1,n2,ic,dn,mn,p,t);$$

or

$$[w1,b1,w2,b2,w3,b3]=cvtrain1(n1,n2,ic,dn,mn,pr,tr,ps,ts);$$

where the training is accomplished by invoking *trainbp* or *trainbpa*, and descriptions of the parameters can be found in their *help* message (see Appendix). The only difference between *cvtrain* and *cvtrain1* is that in the former the testing subset is determined in the algorithm in a random manner and takes about 20 percent of the training set, while in the latter they are provided beforehand as *pr*, *tr*, *ps*, *ts*.

Several calculations have been carried out for two problems. The first problem is the example problem used in 5.5, now with 21 points as training subset and 20 points as testing subset, and the noise level still being at $\sigma = 0.15$. The second is that in the ultimate moment capacity problem (5.4.2.1), with the 21 sets of original training data as training subset, and the 10 sets of testing data as testing subset. Feed-forward neural networks with different configurations are used and some of the results are displayed in Figs. 25 through 30. It should be noted that for the first problem since there exists a real function, we can have two kinds of errors, one based on the noise polluted training set, another based on points on the real function curve. Since what we want to model is the exact mapping in terms of the real function, and in practical problems this mapping is usually nowhere to be found, the error based on real function should be paid more attention.

Hereafter we look through the results, and some conclusions can be drawn, which are true at least for the specific problems.

- There indeed exists an optimum epoch or range of epoch where the combined error tends to be minimum, as can be seen clearly in Figs. 25(a), 26(a), 27(a) and 30(a), and partly in Figs. 28(a) and 29(a). The optimum epoch for the error based on the real function is less than that for the error based on the training data, the former being about one half of the latter. The optimum epoch number decreases as the network complexity increases.
- As number of the training epoch increases, the error on the training subset, E_{tr} , tends to decrease indefinitely and approaches zero, provided the network neuron number is large enough, as shown in Fig. 26(c), 27(c), 29(c), and 30(b). This results in the ratio E_{ts}/E_{tr} based on training data to increase drastically, as in Fig. 26(b), 27(b), and 29(b). But the ratio E_{ts}/E_{tr} based on the real function is always in the vicinity of 1.0 and very stable (see Fig. 25(b) through 29(b)). The latter is expected since both E_{ts} and E_{tr} here represent the average difference between the network generalization and the real function.
- As to the problem of network complexity, at least for the choices having been made, it seems that the best performance is given by a network having about the same number of neurons as that of the training data sets, a fact mentioned earlier. This observation is based on the assumption that the real mapping is smooth and is unaffected by the noise in the training and testing data.
- Cross-validation technique can be used to investigate a lot of problems in NN training. But it is time-consuming and not mature enough to

be incorporated into a routine training process. A tentative standard for stop to avoid over-training can be $E_{ts}/E_{tr}=2$ and $Epoch \geq mn$ (a minimum training number). But this strategy is not so successful (for instance, it is not clear how to determine mn).

- Over-training is not so serious a problem as it seems to be. At least beyond the optimum epoch the combined error only increases slowly. As to the error ratio of testing over training, it is almost a constant (1.0) if it is based on the real mapping and if the training and testing data are uniformly noise-polluted. This means that training epoch should be high enough. Under-training is worse than over-training and should be avoided.
- From the figures it can be seen that for the example problem the training of network is locally unstable with one hidden layer while not with two hidden layers. From Fig. 30(a) and some other results not included here, we can see the situation for the ultimate moment capacity problem in 5.4.2.1 is the same. Obviously, in some way networks with two hidden layers are superior to those with only one (Ref. 1).
- If a testing data is incorporated into a cross-validation process to train a network, it contributes (at least indirectly) to the training and loses its status as independent testing data.

8. Methods of Obtaining Continuum Models

A lot of methods have been used to develop continuum models to represent com-

plex structures. Many of these methods involve the determination of the appropriate relationships between the geometric and material properties of the original structure and its continuum models. An important observation is that the continuum model is not unique, and determining the continuum model for a given complex structure is inherently ill-posed therefore diverse approaches can be used. This can be clearly shown in the following example of determining continuum models for a lattice structure.

The single-bay double-laced lattice structure shown in Fig. 31 has been studied in Ref. 12, 13 and 14 with different approaches to the continuum modeling. This lattice structure with repeating cells can be modeled by a continuum beam if the beam's properties is properly provided.

Noor et al's method include the following steps (Ref. 12): (1)introducing assumptions regarding the variation of the displacements and temperature in the plane of the cross section for the beamlike lattice, (2)expressing the strains in the individual elements in terms of the strain components in the assumed coordinate directions, (3)expanding each of the strain components in a Taylor series, and (4)summing up the thermoelastic strain energy of the repeating elements which then gives the thermoelastic and dynamic coefficients for the beam model in terms of material properties and geometry of the original lattice structure.

In Sun et al (Ref.13), the properties of the continuum model is obtained respectively by relating the deformation of the repeating cell to different load settings under specified boundary conditions. For example, the shear rigidity GA is obtained by performing a numerical shear test in which a unit shear force is applied at one end of the repeating cell and the corresponding shear deformation is calculated by using a finite element program. The mass and rotatory inertia are calculated with a averaging procedure.

Lee put forward a method that he thought to be more straightforward (Ref. 14). He used an extended Timishenko beam to model the equivalent continuum beam. By expressing the total strain and kinetic energy of the repeating cell in terms of the displacement vector at both ends of the continuum model, and equating them to those obtained through the extended Timishenko beam theory, he got a group of relations. The number of these relations, $2N(1+2N)$, where N is the degree of freedom of the continuum model, is usually larger than that of the equivalent continuum beam properties to be determined. Lee then introduced a procedure in which the stiffness and mass matrices for both the lattice cell and the continuum model are reduced and so is the number of relations. Yet how to reduce the number of relations to be equal to the number of unknowns seems depend on luck.

All the above three methods give close results for the continuum model properties, and the continuum models also generate promising results for the lattice structure.

9. An Example of NN Modeling of Continuum Models

Emphasizing the application of NN, we choose an approach similar to that in Ref. 13, that is, to derive the properties of the beam by investigating the force-deformation relationships of the repeating cell in certain boundary conditions. The approach is illustrated in Figs. 32 (a), (b) and (c), where the beam's axial rigidity EA , bending rigidity EI , and shearing rigidity GA are calculated respectively by using the results of finite element analysis of the repeating cell in different load conditions. Concerning the finite element analysis of 3-D lattice structures one can consult Ref. 15.

There are five parameters of the repeating cell for the lattice structure in Fig.

31 that can be varied, the longitudinal bar length L_c , the batten length L_g , and the longitudinal, batten and diagonal bar area, A_c , A_g and A_d . Generally, a function with more variables will be more complex and it will be more difficult for a neural network to simulate its performance. A NN with more input variables needs much more training data since in the training data each variable should vary separately. As can be shown in the following, this kind of "coarse" training data pose an obstacle to most of the training algorithms.

Three scenarios were investigated, with the number of input variables set to be 2, 3 and 4 respectively.

9.1 Neural Network with 2 Input Variables

The input variables are L_c and A_c . The number of training data sets is $400=20 \times 20$. The number of testing data, most of which are located at centers among the training data mesh, is also $400=20 \times 20$. Part of the results, about GA, is shown in Fig. 33. Simulations on the testing data and the relative errors of a FF NN (2-10-1) trained with Levenberg-Marquardt (*trainlm*) are shown in Figs. 33 (b) and (c). Results of a RBF NN doing the same job are shown in Figs. 33 (d) and (e).

We could not say that FF is superior over RBF just because its testing accuracy shown in Fig. 33 (c) is higher than RBF's shown in Fig. 33 (e). We can adjust the training criterion to change a RBF NN's behavior and it should be noted that what is shown in Figs. 33 (d) and (e) are not RBF NN's best performances.

9.2 Neural Network with 3 Input Variables

The input variables were chosen as L_c , A_c and A_d . The number of training data sets is $343=7\times7\times7$. For this case, the effectiveness of different training algorithms can be seen clearly in Fig. 34. When ordinary back-propagation training algorithm, i.e. *trainbp* is used, it is very hard to train the NN to the error level of 10^{-1} , as shown in Fig. 34 (a). When the adaptive learning technique is included, an improvement can be made, but it is still hard to reach the 10^{-2} error level, as can be seen in Fig. 34 (b). Now if the algorithm with Levenberg-Marquardt (*trainlm*) is used, it is quite easy to push the training error level to the order of 10^{-4} .

The improvement by *trainlm* is really amazing. All the training algorithms carry out an optimization process. While *trainbp* uses steepest-descent method with constant step size, *trainbpa* accelerates the process by adjusting the step size. On the other hand, *trainlm* adopts a kind of modified Newton's Methods, which adjusts both the searching direction and the step size. Concerning the optimization methods one can consult Ref. 16.

Samples of the NN simulation results are given in Table 3, where the desired values and values obtained by Noor et al (Ref. 12) and Lee (Ref. 14) are also presented.

9.3 Neural Network with 4 Input Variables

The input variables were chosen as L_c , A_c , A_g and A_d . The number of training data sets is $625=5\times5\times5\times5$. For this case only *trainlm* could train a FF NN that could give resonable results. Samples of the NN simulation results are given in Table 4.

10. References

- [1]L. P. J. Veelenturf, *Analysis and Application of Artificial Neural Networks*, Prentice Hall, 1995
- [2]S. Haykin, *Neural Networks: A Comprehensive Foundation*, Macmillan College Publishing Company, 1994
- [3]G. Cybenko, "Neural Networks in Computational Science and Engineering", *IEEE Computational Science and Engineering*, Spring 1996
- [4]K. Hornik, M. Stinchcombe and H. White, "Multilayer Feedforward Networks are Universal Approximators", *Neural Networks*, Vol. 2, 359-366, 1989
- [5]E. J. Hartman, J. D. Keeler and J. M. Kowaski, "Layered Neural Networks with Gaussian Hidden Units as Universal Approximators", *Neural Computation*, Vol. 2, 210-215, 1990
- [6]L. C. Rabelo, V. Shastri, E. Onyejekwe, and J. Vila, "Using Neural Networks and Abductive Modeling for Device-Independent Color Correction", *Scientific Computing and Automation*, May 1996
- [7]K. M. Abdalla, G. E. Stavroulakis, "A Backpropagation Neural Network Model for Semi-rigid Steel Connections", *Microcomputers in Civil Engineering*, 10 (1995) 77-87
- [8]R. D. Vanluchene, R. Sun, "Neural Network in Structural Engineering", *Microcomputers in Civil Engineering*, 5 (1990) 207-215
- [9]D. J. Gunaratnam, J. S. Gero, "Effect of Representation on the Performance of Neural Networks in Structural Engineering Applications", *Microcomputers in Civil*

- [10]S. L. Lipson, "Single-Angle and Single-Plate Beam Framing Connections", *Proceedings of Canadian Structure Engineering Conference*, Toronto, Ontario 1968, 141-162
- [11]Mark J. L. Orr, "MATLAB Routines for Subset Selection and Ridge Regression in Linear Neural Networks", Center for Cognitive Science Edinburg University, Scotland, UK.
Accessible at <http://www.cns.ed.ac.uk/people/mark/rbf.html>
- [12]A. K. Noor, M. S. Anderson and W. H. Greene, "Continuum Models for Beam- and Platelike Lattice Structures", *AIAA Journal*, Vol. 16, No. 12, Dec. 1978, pp. 1219-1228
- [13]C. T. Sun, B. J. Kim and J. L. Bogdanoff, "On the Derivation of Equivalent Simple Models for Beam- and Plate-Like Structures in Dynamic Analysis", AIAA Paper 81-0624, 1981, pp. 523-532
- [14]U. Lee, "Dynamic Continuum Modeling of Beamlike Space Structures Using Finite-Element Matrices", *AIAA Journal*, Vol. 28, No. 4, Apri. 1990, pp. 725-731
- [15]C. T. F. Ross, *Finite Element Methods in Structural Mechanics*, Ellis Horwood Ltd., 1985
- [16]J.-S. R. Jang, C.-T. Sun and E. Mizutani, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice Hall, Inc., 1997

Appendix Help messages for MATLAB Algorithms *cvtrain* and *cvtrain1*

```
>> help cvtrain
```

```
[w1,b1,w2,b2,w3,b3]=cvtrain(n1,n2,ic,dn,mn,p,t)
```

CROSS-VALIDATION TRAINING OF FEED-FORWARD NETWORK BY BACK-PROPAGATION

Input Parameters:

- n1 - Neuron number of the first hidden layer.
- n2 - Neuron number of the second hidden layer (if no input 0).
- ic - 1: without adaptive learning. 2: with adaptive learning.
- dn - Number of training for each batch.
- mn - Minimum training number.
- p - m by n matrix of input vectors.
- t - 1 by n matrix of target vectors.
- m - Number of input variables.
- n - Number of training data sets.
- l - Number of output variables.

Return Parameters:

- wi - network weights.
- bi - network biases.

```
> help cvtrain1
```

```
[w1,b1,w2,b2,w3,b3]=cvtrain1(n1,n2,ic,dn,mn,pr,tr,ps,ts)  
or (with pre-training)
```

```
[w1,b1,w2,b2,w3,b3]=cvtrain1(n1,n2,ic,dn,mn,pr,tr,ps,ts,w1,b1,w2,b2,w3,b3)
```

CROSS-VALIDATION TRAINING OF FEED-FORWARD NETWORK BY BACK-PROPAGATION

Input Parameters:

- n1 - Neuron number of the first hidden layer.
- n2 - Neuron number of the second hidden layer (if no input 0).
- ic - 1: without adaptive learning. 2: with adaptive learning.
- dn - Number of training for each batch.
- mn - Minimum training number.
- pr - m by nr matrix of input vectors for training.
- tr - 1 by nr matrix of target vectors for training.
- ps - m by ns matrix of input vectors for testing.
- ts - 1 by ns matrix of target vectors for testing.
- m - Number of input variables.
- nr - Number of training data sets.
- ns - Number of testing data sets.
- l - Number of output variables.
- wi - network weights (for cases with pre-training).
- bi - network biases (for cases with pre-training).

Return Parameters:

- wi - network weights.
- bi - network biases.

Table 1 Single Angle Beam-to Column Connections Bolted to Both Beam and Column: Experiments from Lipson [7,10]

Experiment Number	Number of Bolts	Angle Thickness (inch)	Angle Length (inch)
1	2	.25	5.5
2	3	.25	8.5
3	4	.25	11.5
4	6	.25	14.5
5	6	.25	17.5
6	4	.3125	11.5

Table 2 Training Data for a Simple Beam Problem

Location	Input (Moment)										Output (location)
	0	1/9	2/9	3/9	4/9	5/9	6/9	7/9	8/9	1	
1	0	.4	.35	.3	.25	.2	.15	.1	.05	0	1/9
2	0	.3	.6	.9	.75	.6	.45	.3	.15	0	3/9
3	0	.2	.4	.6	.8	1.0	.75	.5	.25	0	4/9
4	0	.1	.2	.3	.4	.5	.6	.7	.35	0	7/9

Table 3 Comparison of Continuum Model Properties for a Lattice Repeating Cell

Features of the cell: $E = 7.17 \times 10^{10} \text{ N/m}^2$, $L_g = 5.0 \text{ m}$, $A_g = 6 \times 10^{-5} \text{ m}^2$
 Variables specified: $L_c = 7.5 \text{ m}$, $A_c = 8 \times 10^{-5} \text{ m}^2$, $A_d = 4 \times 10^{-5} \text{ m}^2$

	present	present (FF simulated, trained by <i>trainbpa</i> , 3 variables)	present (RBF simulated, 3 variables)	present (FF simulated, trained by <i>trainln</i> , 3 variables)	U. Lee	Noor et al
EA (10^7 N)	2.659	2.60	2.66	2.656	2.71	2.53
GA (10^6 N)	2.183	2.24	2.17	2.186	2.2	2.2
EI ($10^7 \text{ N} \cdot \text{m}^2$)	8.147	8.08	8.17	8.148	8.20	8.01

Table 4 Comparison of Continuum Model Properties for a Lattice Repeating Cell

Features of the cell: $E = 7.17 \times 10^{10} \text{ N/m}^2$, $L_g = 5.0 \text{ m}$
Variables specified: $L_c = 7.5 \text{ m}$, $A_c = 8 \times 10^{-5} \text{ m}^2$, $A_d = 4 \times 10^{-5} \text{ m}^2$, $A_g = 6 \times 10^{-5} \text{ m}^2$

	present	present (FF simulated, trained by <i>train</i> , 4 variables)	U. Lee	Noor et al
EA ($10^4 N$)	2.659	2.682	2.71	2.53
GA ($10^6 N$)	2.183	2.134	2.2	2.2
EI ($10^4 N \cdot m^2$)	8.147	8.181	8.20	8.01

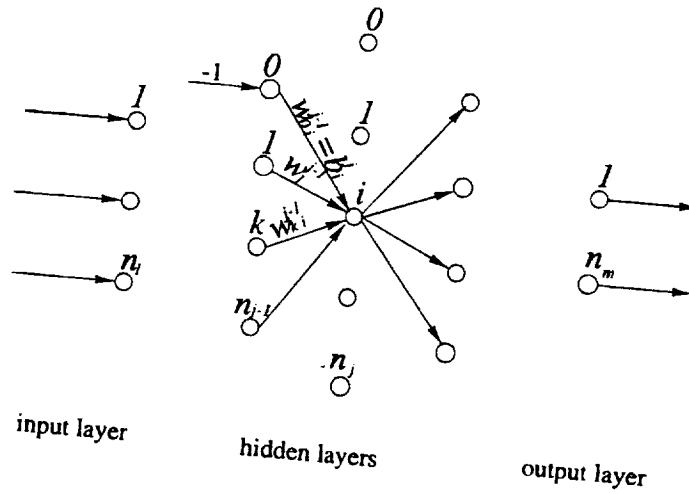


Fig. 1(a) Feed-Forward Multi-Layer Neural Network

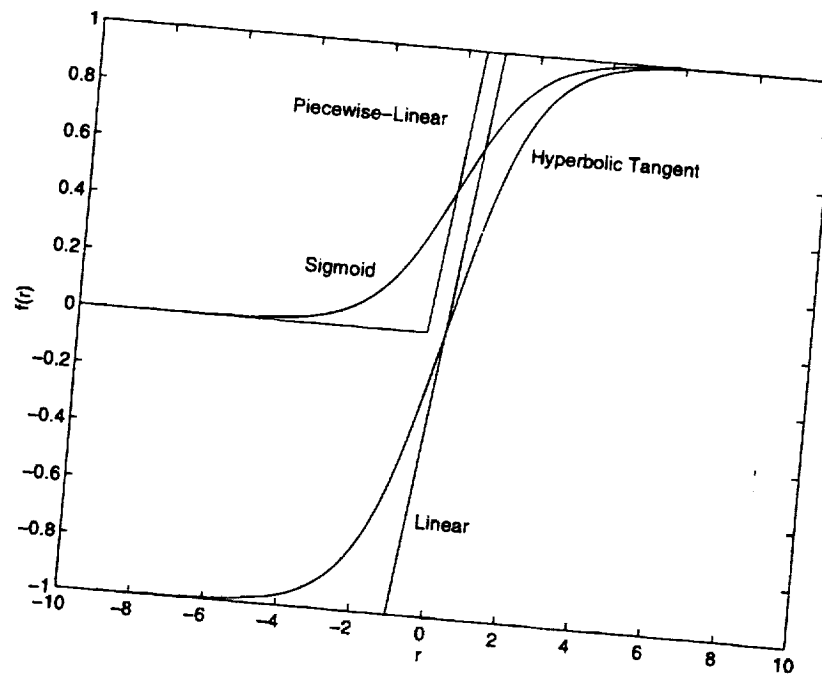
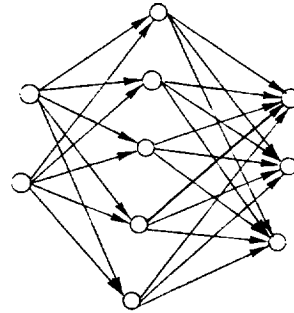


Fig. 1(b) Transfer Functions



number of
neurons: n_1 n_2 n_3
input hidden output
layer layer layer

Fig. 2 Radial Basis Function Neural Network

Fig. 3(a) The influence of initial weights and biases: training histories of two cases where everything kept the same except the initial parameters

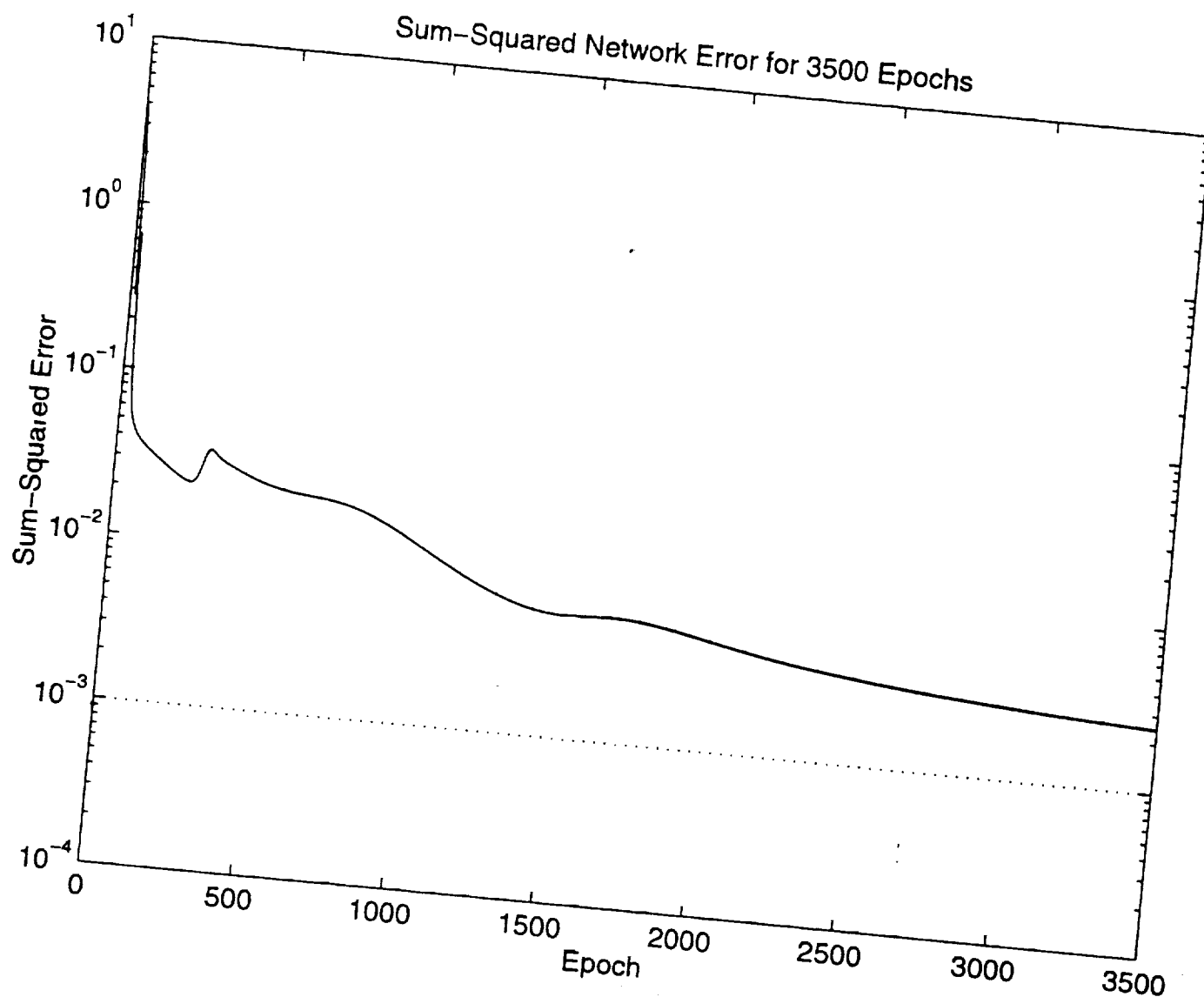


Fig. 3(b) The influence of initial weights and biases: training histories of two cases where everything kept the same except the initial parameters

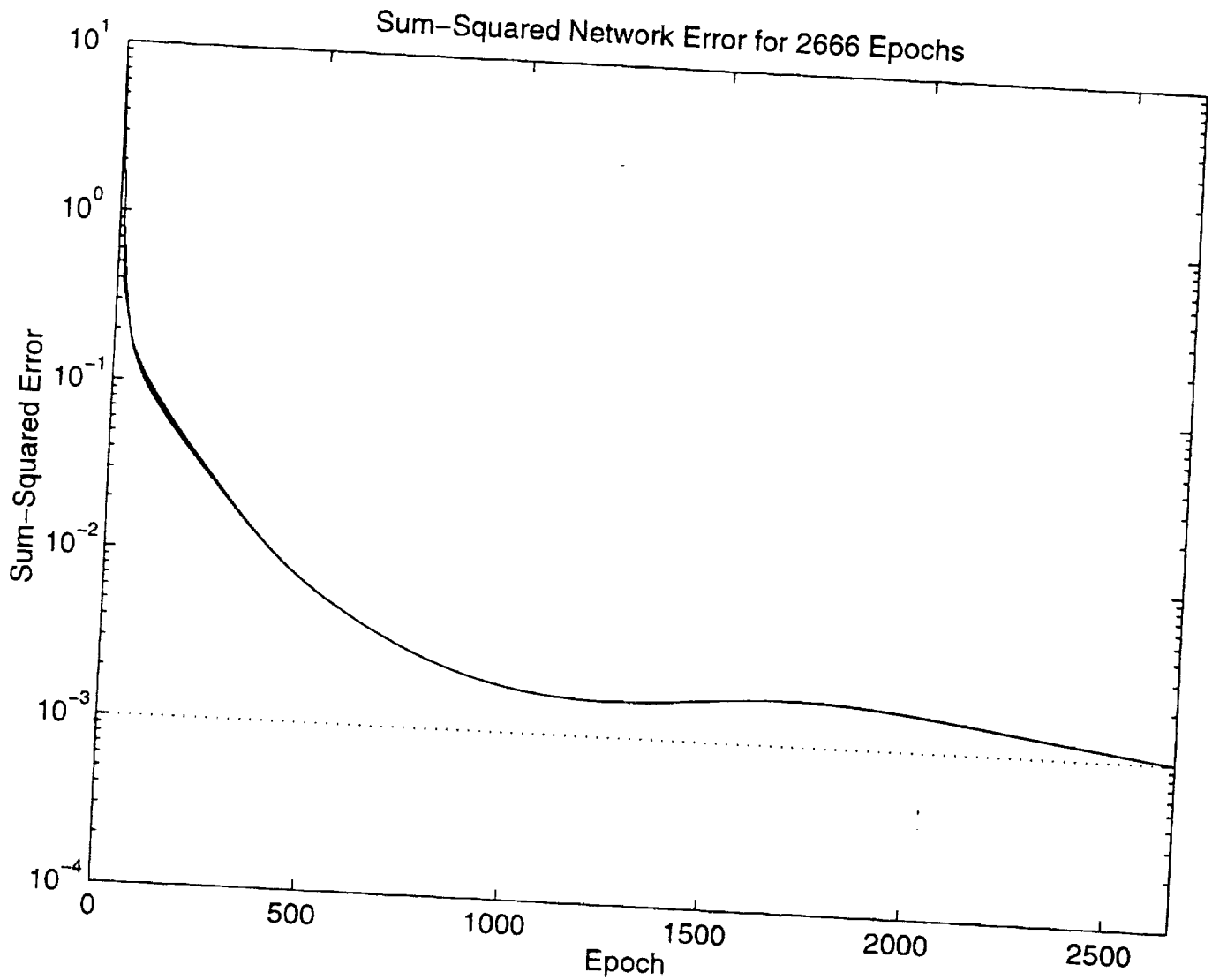


Fig.4 (a) 1-9-1 feed-forward NN trained with back-propagation:
comparison between simulation (solid line) and desired value ('+')

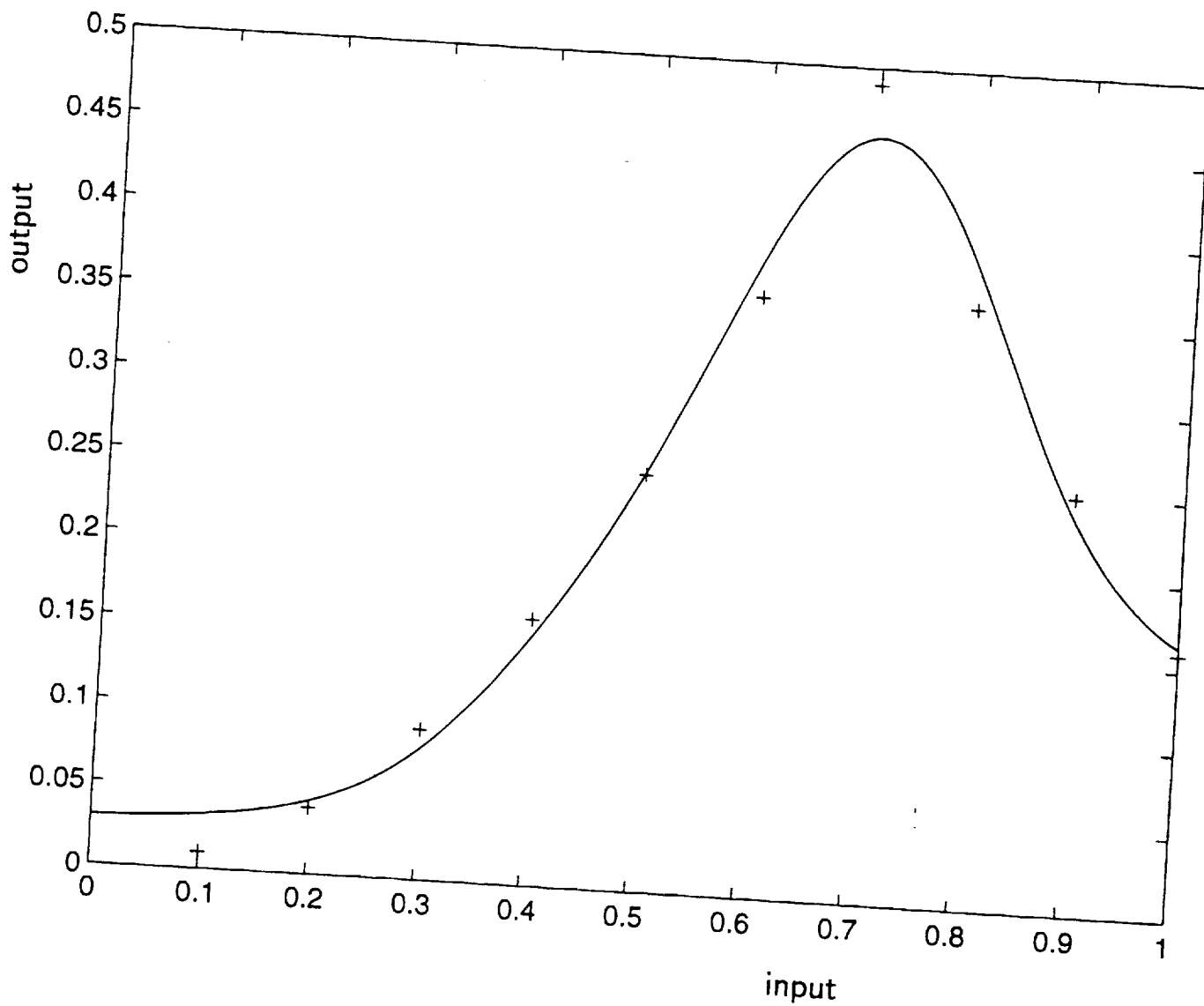


Fig.4 (b) 1-9-1 feed-forward NN trained with back-propagation:
training history

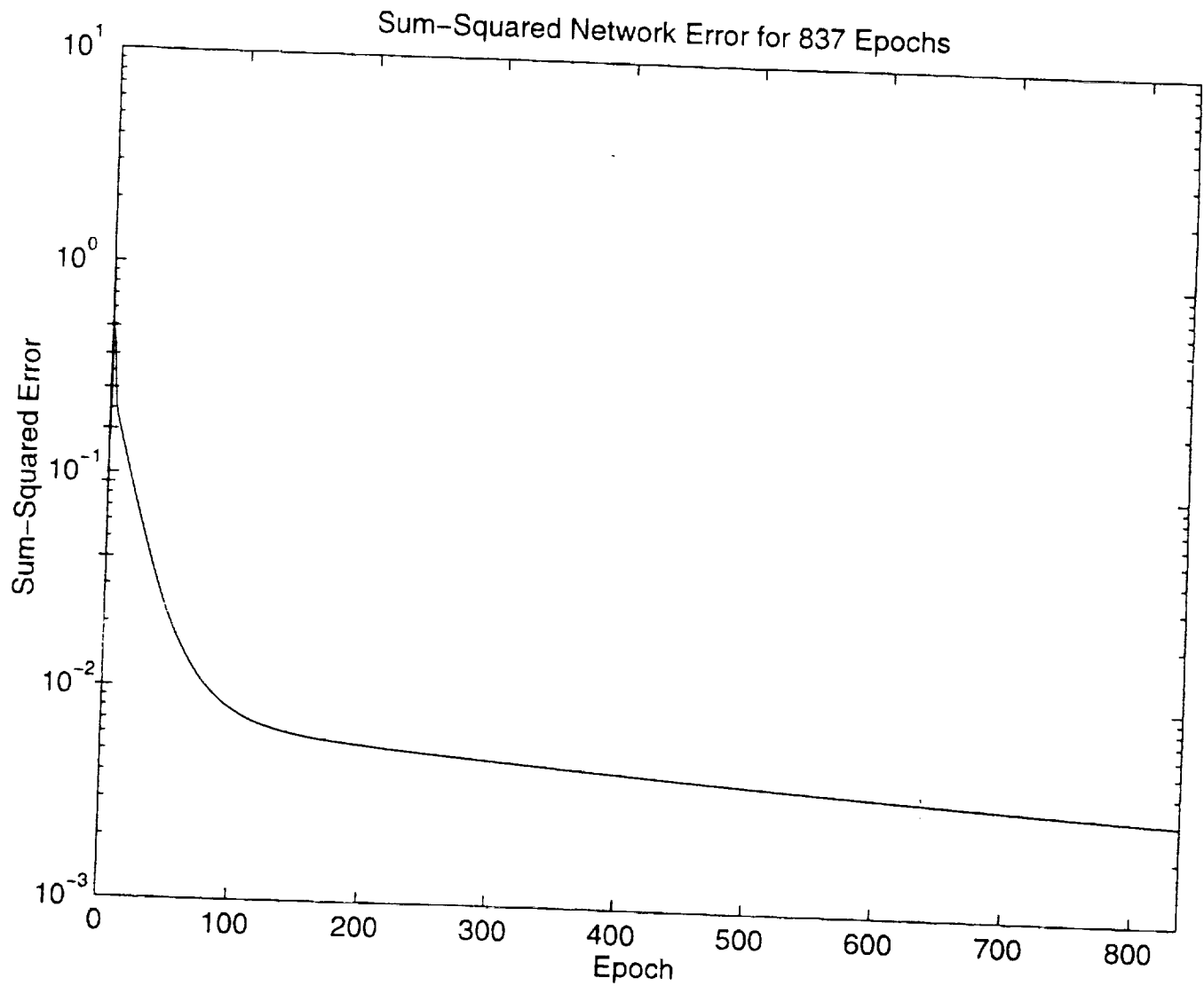


Fig.5 (a) 1-9-6-1 feed-forward NN trained with back-propagation:
comparison between simulation (solid line) and desired value ('+')

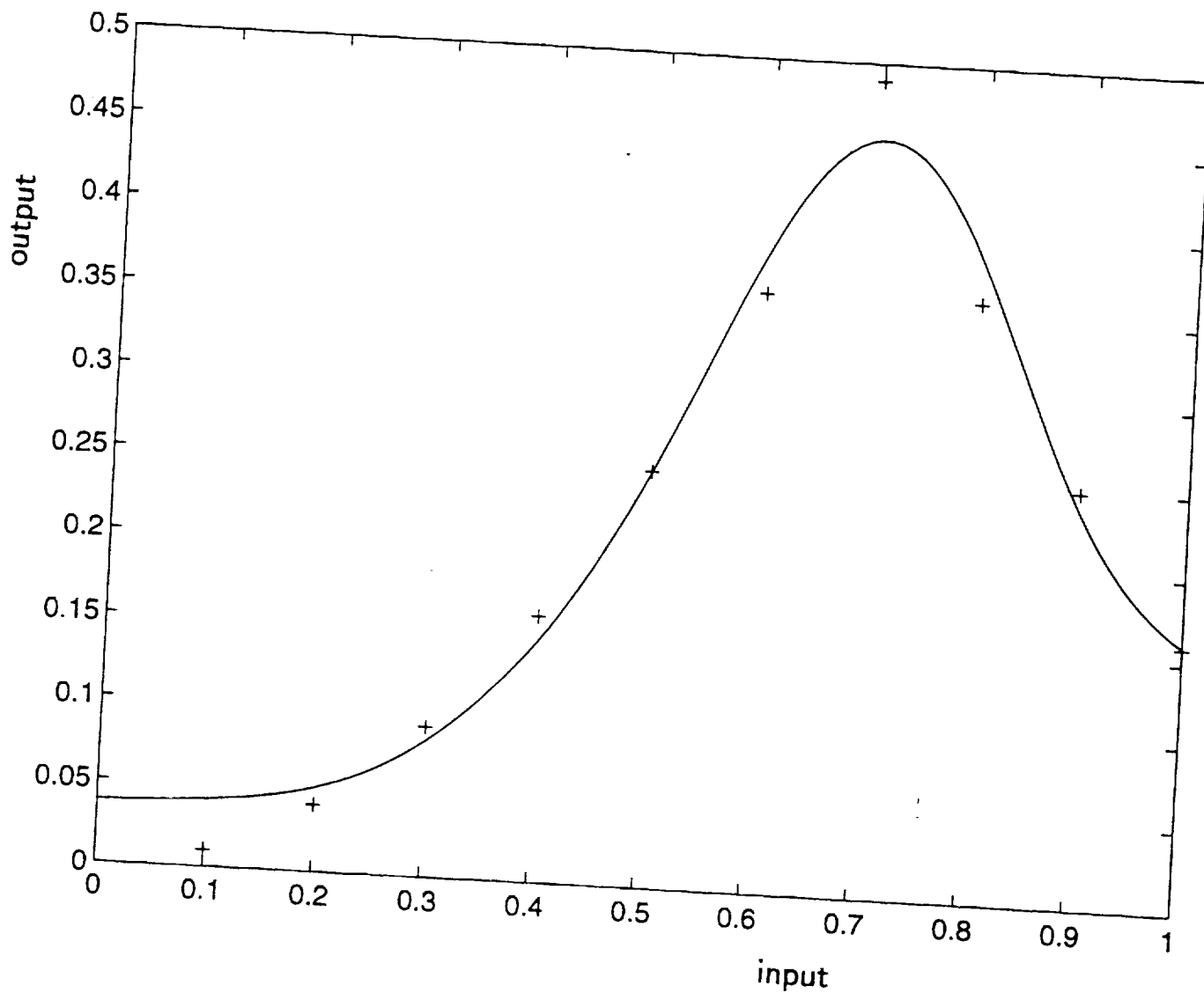


Fig.5 (b) 1-9-6-1 feed-forward NN trained with back-propagation:
training history

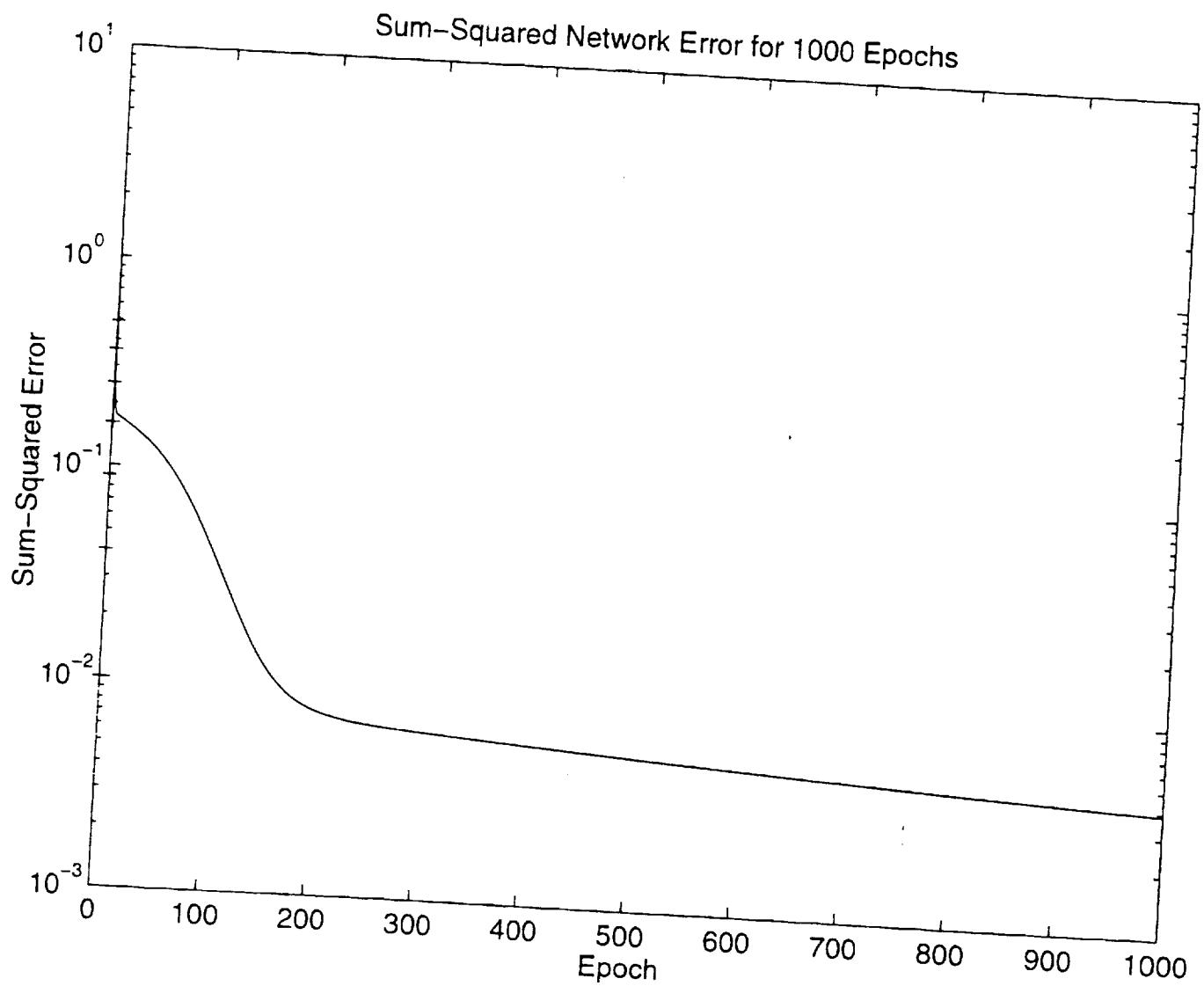


Fig.6 (a) 1-9-6-1 feed-forward NN trained with BP and adaptive learning:
comparison between simulation (solid line) and desired value ('+')

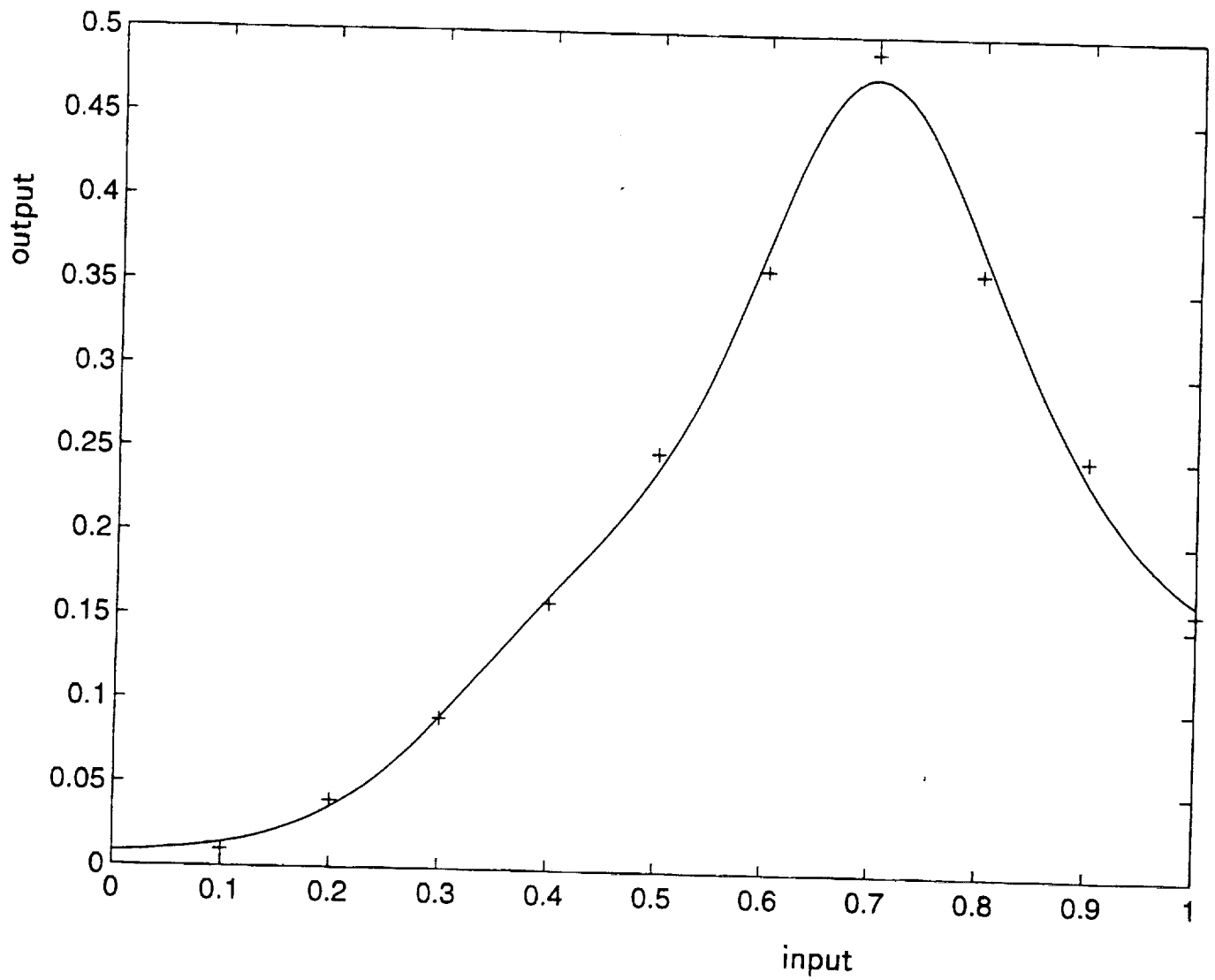
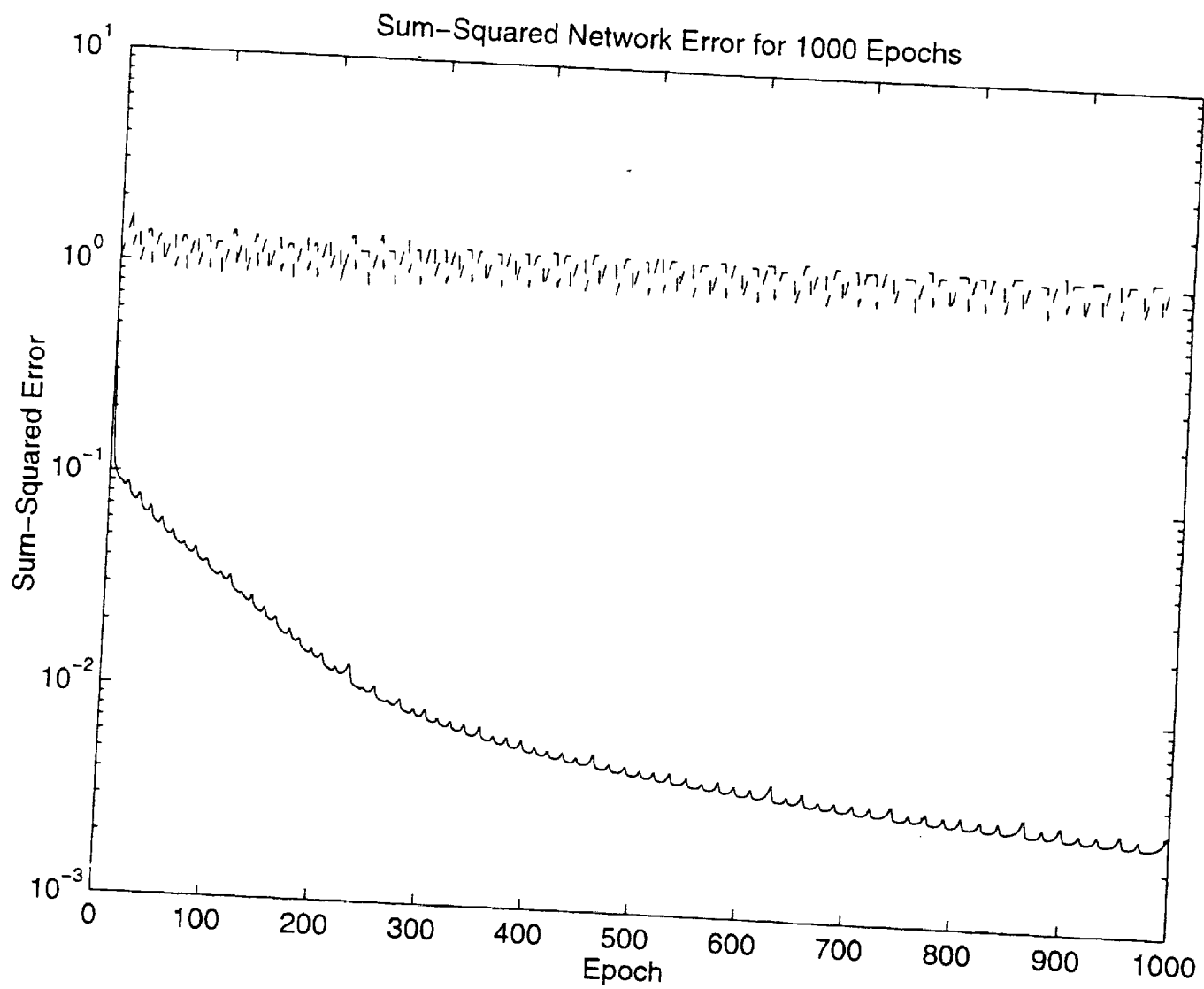


Fig.6 (b) 1-9-6-1 feed-forward NN trained with BP and adaptive learning training history



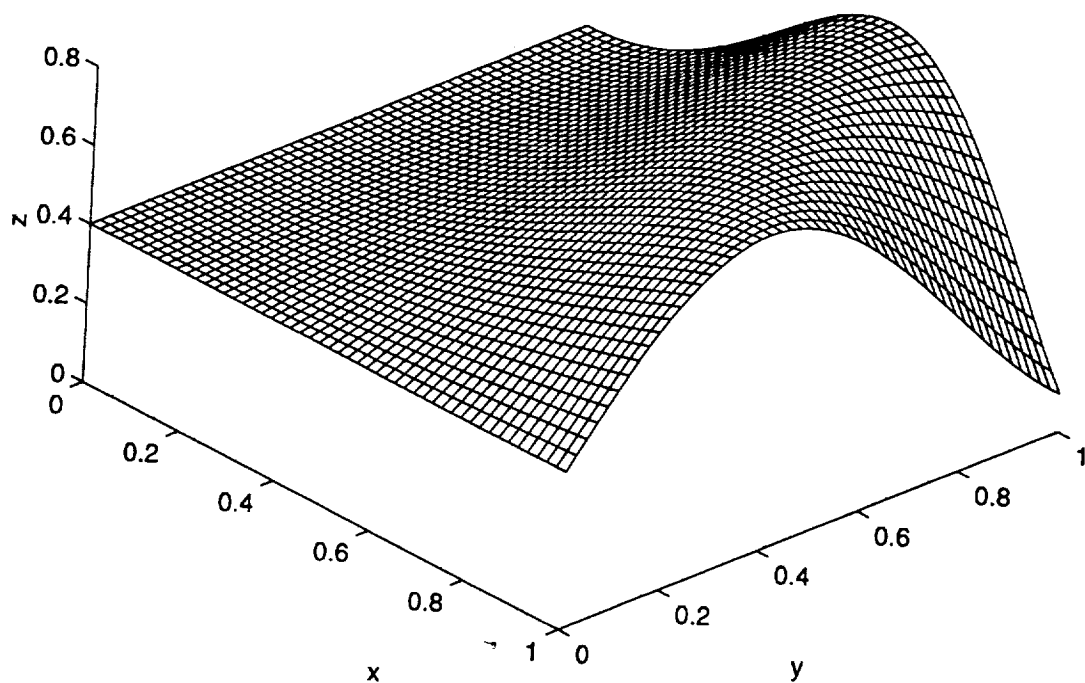


Fig. 7 The mapping of $z = \frac{2}{5}(1 + \sin(4xy^2))$

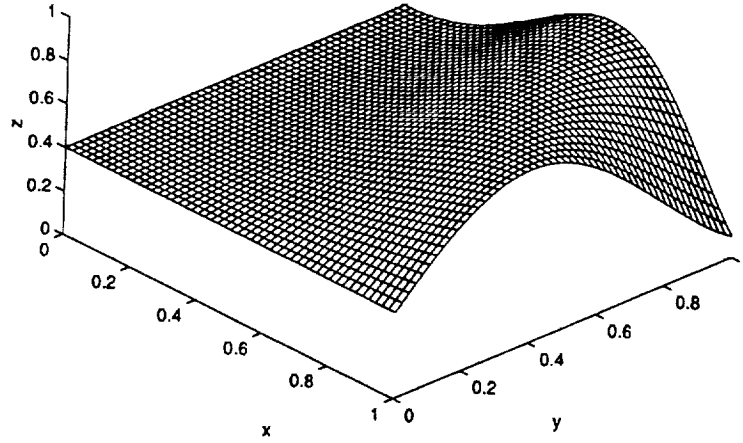


Fig. 8(a) Simulated mapping by a 2-10-1 NN trained by 11×11 uniformly distributed data

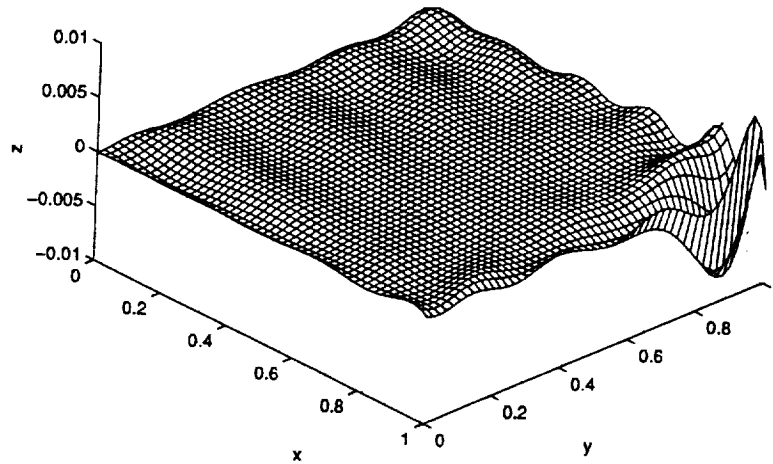


Fig. 8(b) The relative error compared to the exact mapping (Fig. 7)

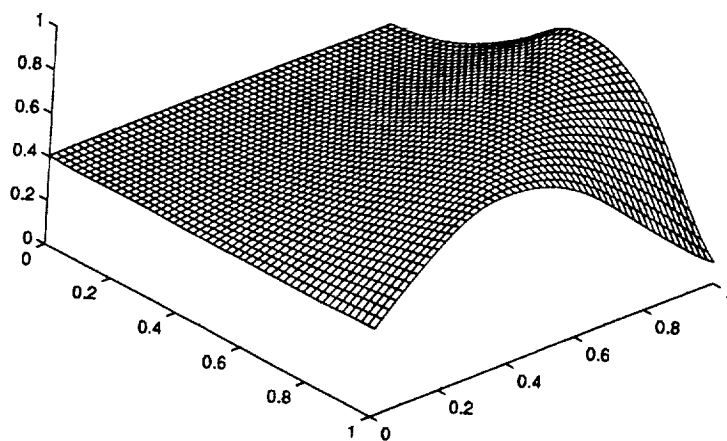


Fig. 9(a) Simulated mapping by a 2-10-1 NN trained by 121 sets of randomly distributed data

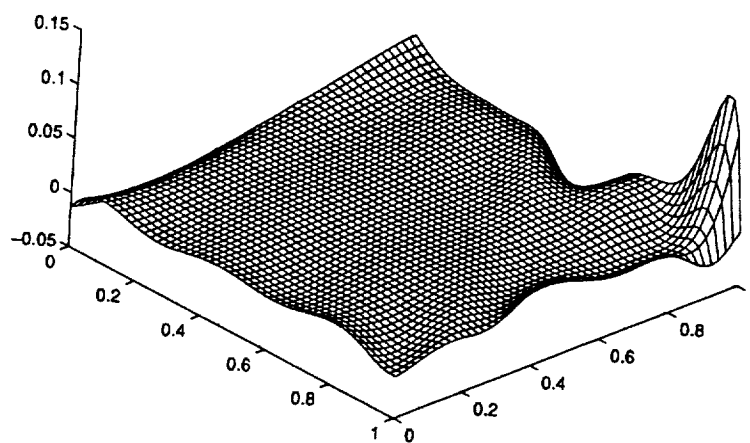


Fig. 9(b) The relative error compared to the exact mapping (Fig. 7)

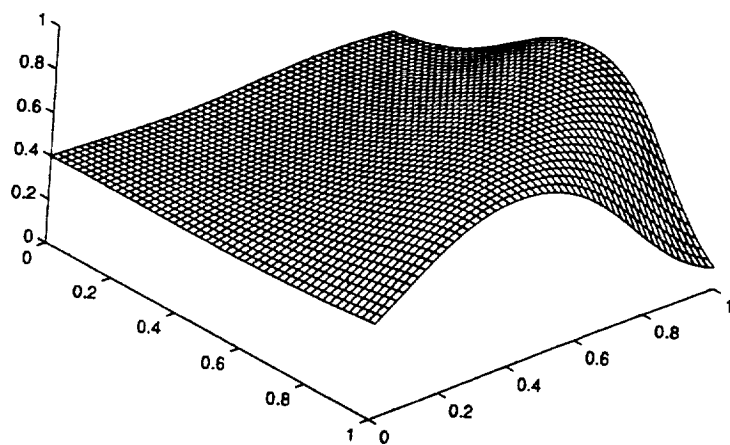


Fig. 10(a) Simulated mapping by a 2-10-1 NN trained by 24 sets of randomly distributed data

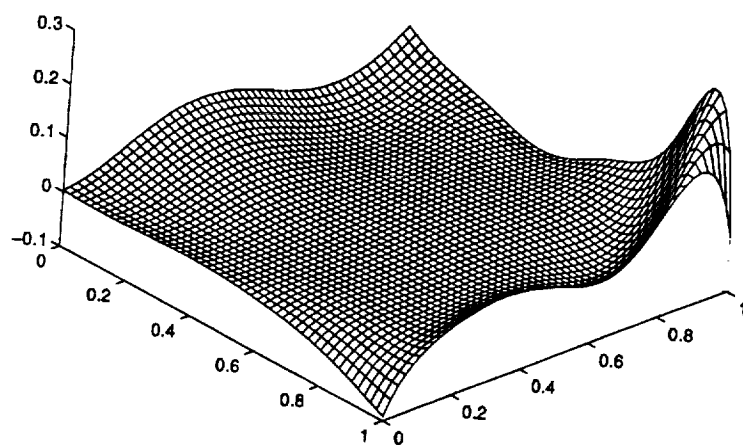


Fig. 10(b) The relative error compared to the exact mapping (Fig. 7)

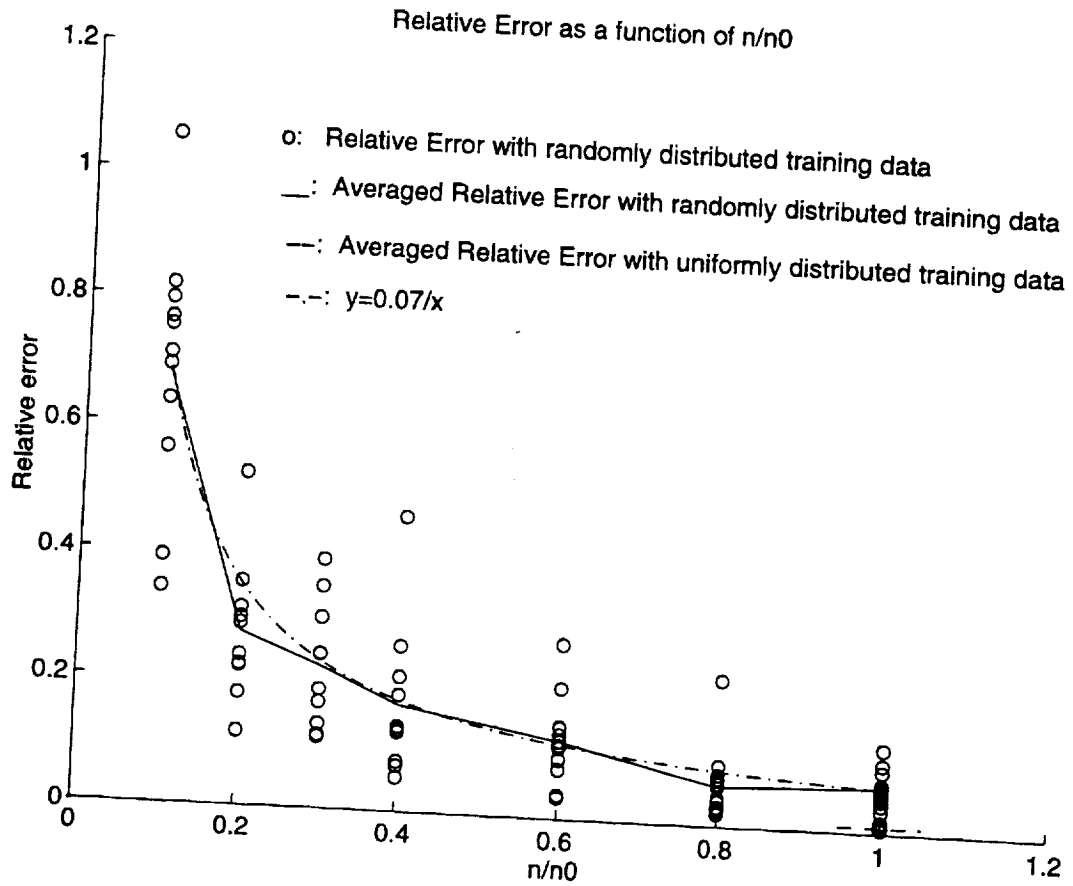


Fig. 11 The relative error as a function of $\eta = \frac{n}{n_0}$

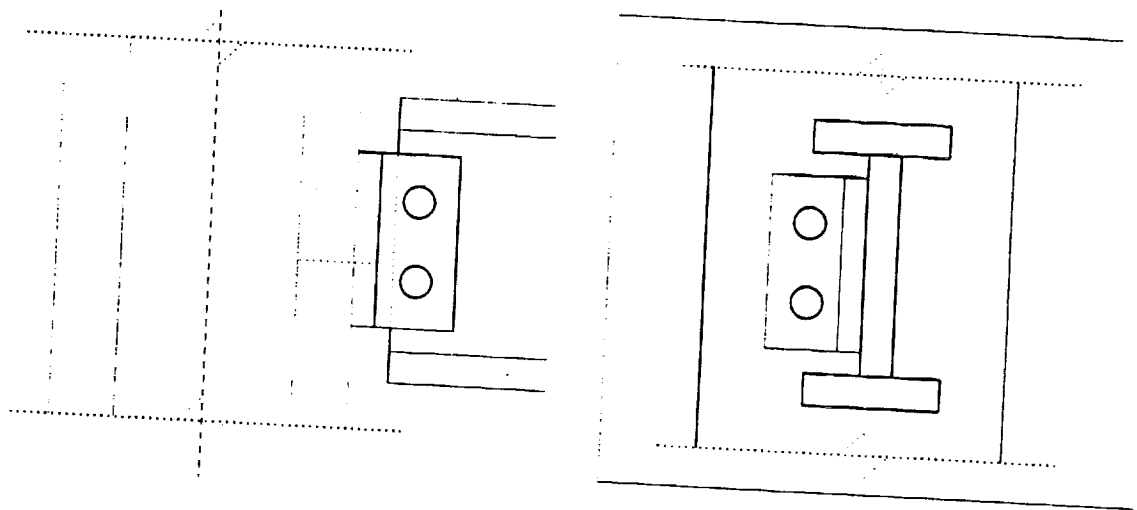


Fig. 12 Single Angle Beam-to Column Connection

Fig. 13(a) Reproducing results of [7] by a 3-50-50-22 NN: Comparison of experimental results (lines) and simulation ('o': training, '+' : testing)

Exp. 1.2.3.5.6 for training. 'logsig' for *initff*

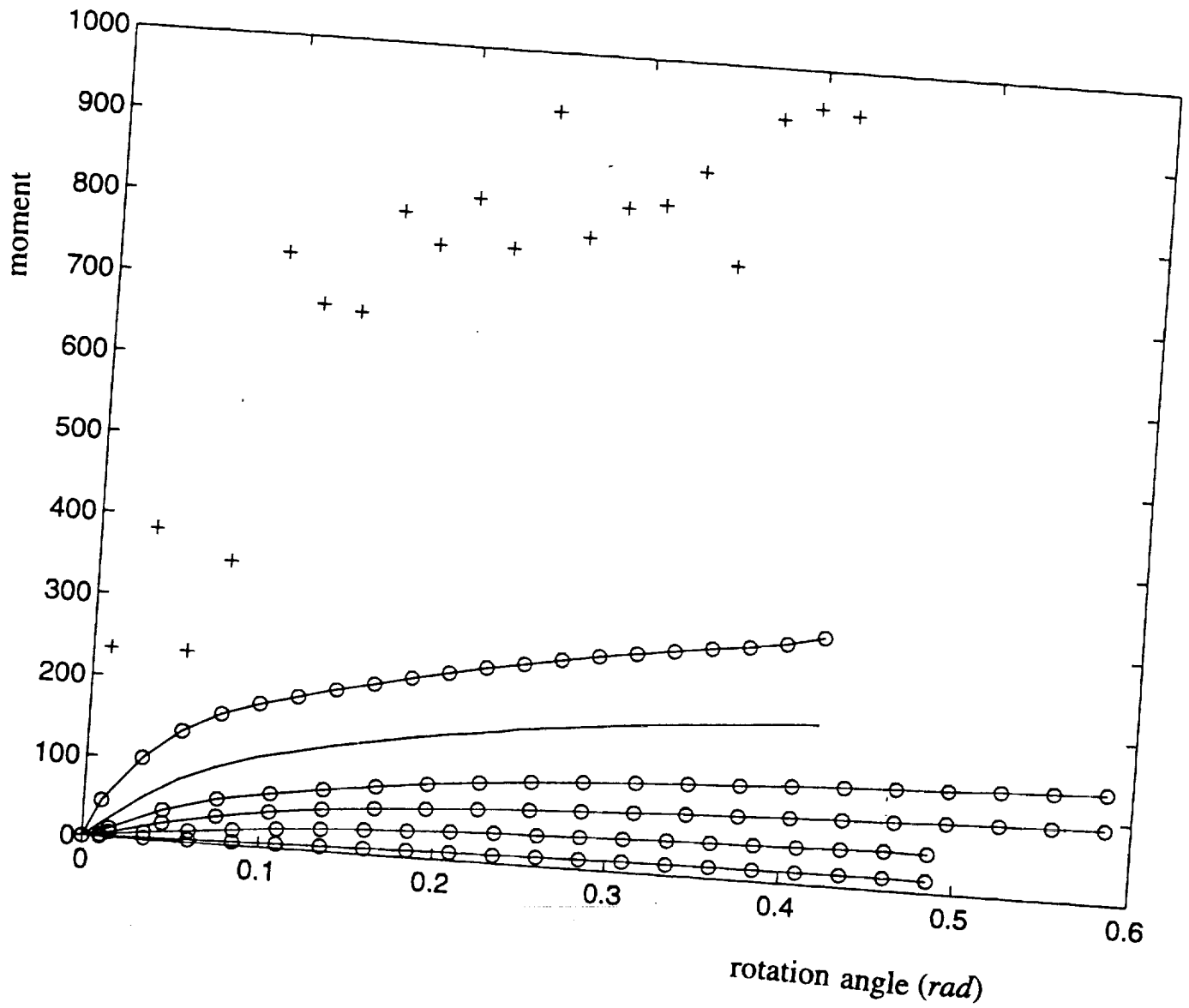


Fig. 13(b) Reproducing results of [7] by a 3-50-50-22 NN: Comparison of experimental results (lines) and simulation ('o': training, '+' : testing)

Exp. 1.2,3,4,6 for training, 'logsig' for initff

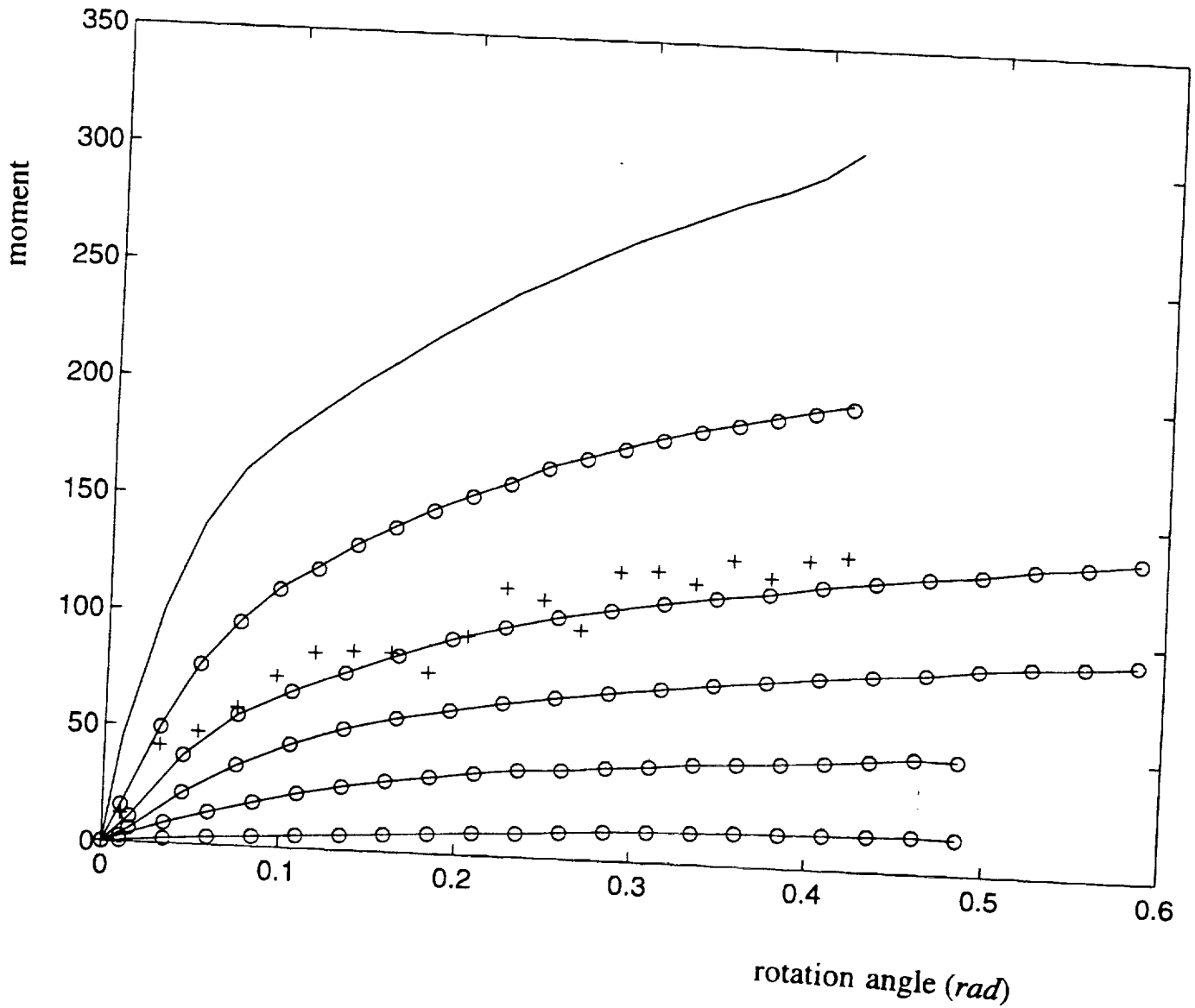


Fig. 13(c) Reproducing results of [7] by a 3-50-50-22 NN: Comparison of experimental results (lines) and simulation ('o': training, '+' : testing)

Exp. 1.2.3.5.6 for training, 'tansig' for initff

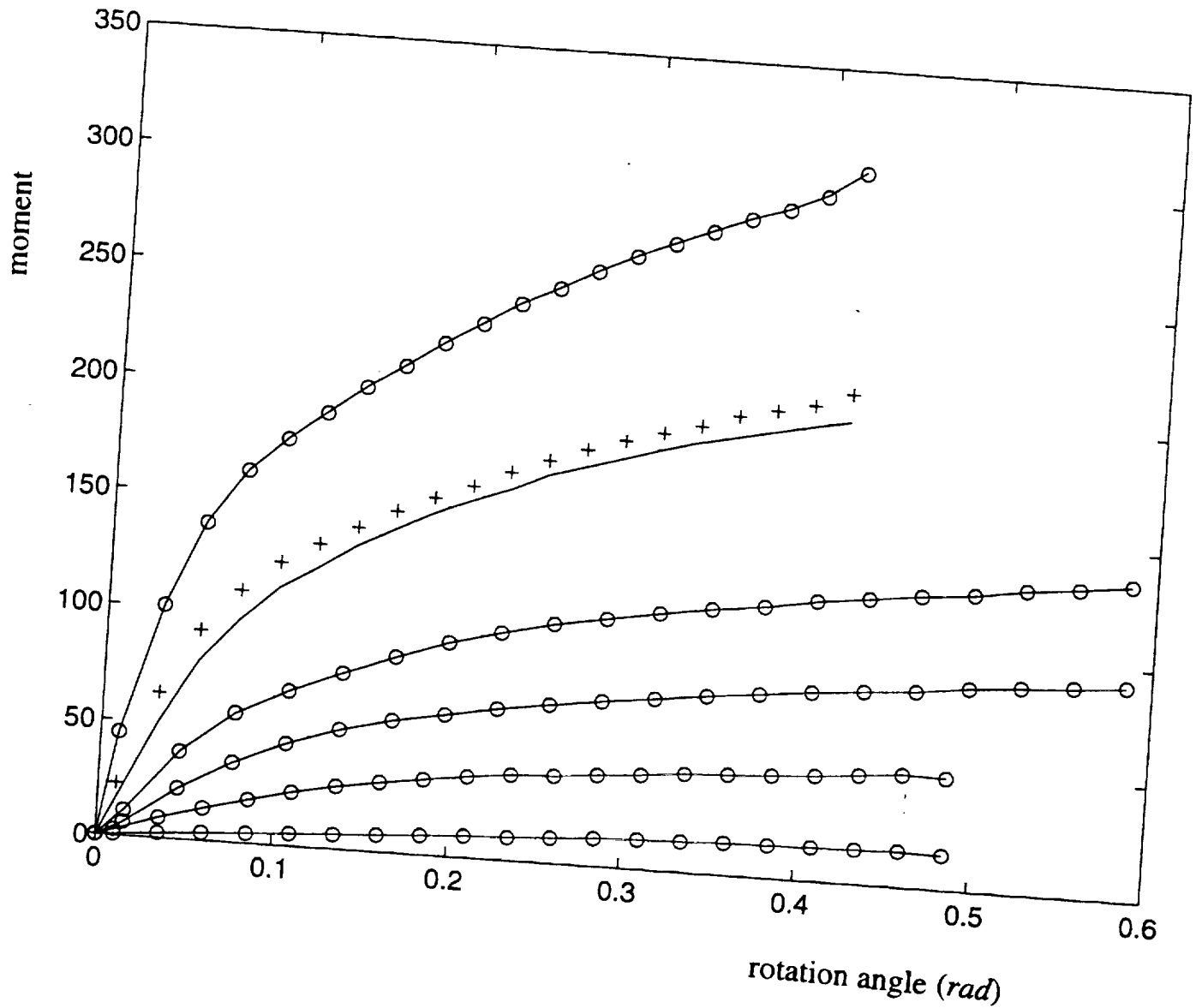


Fig. 14

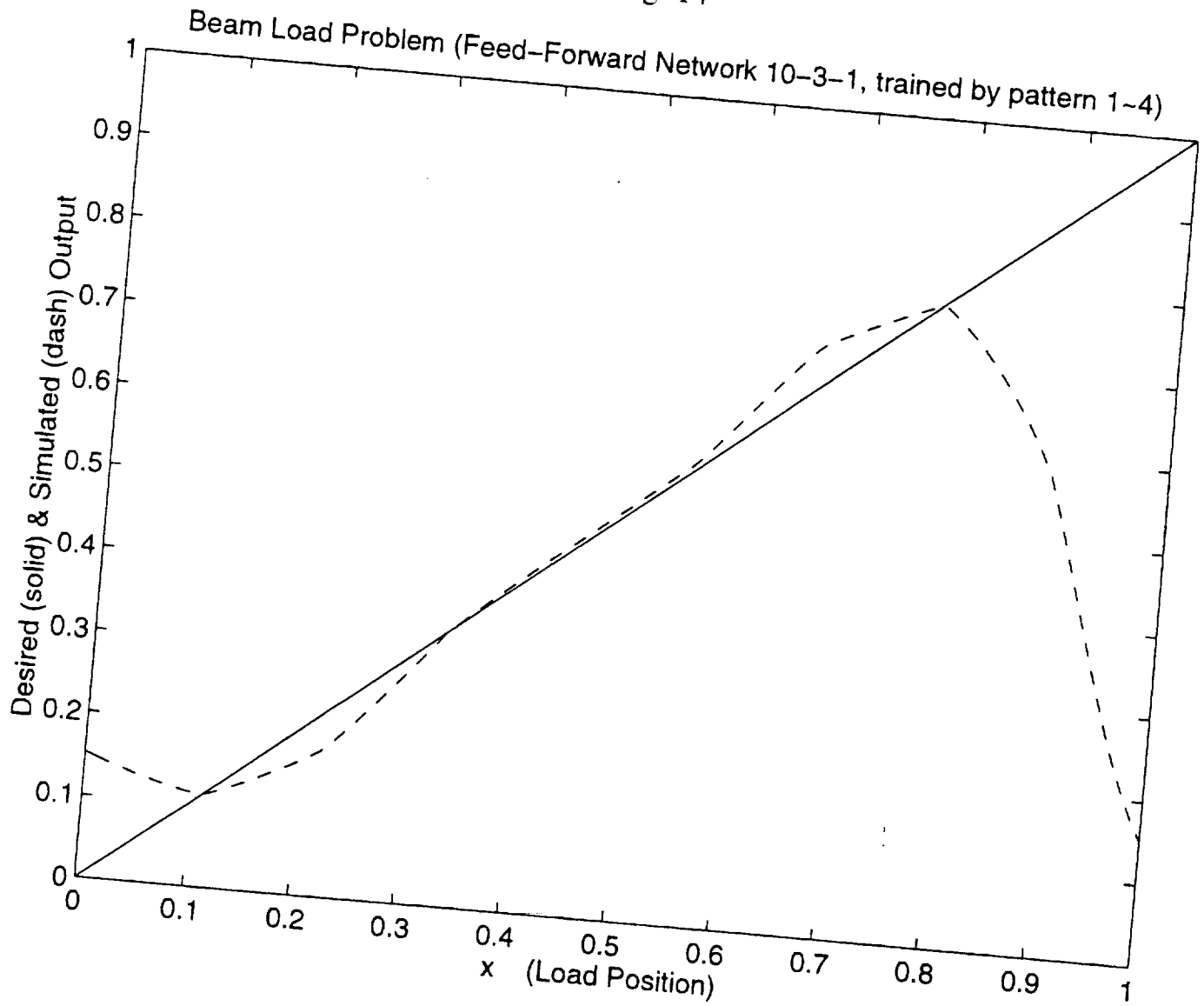


Fig.15

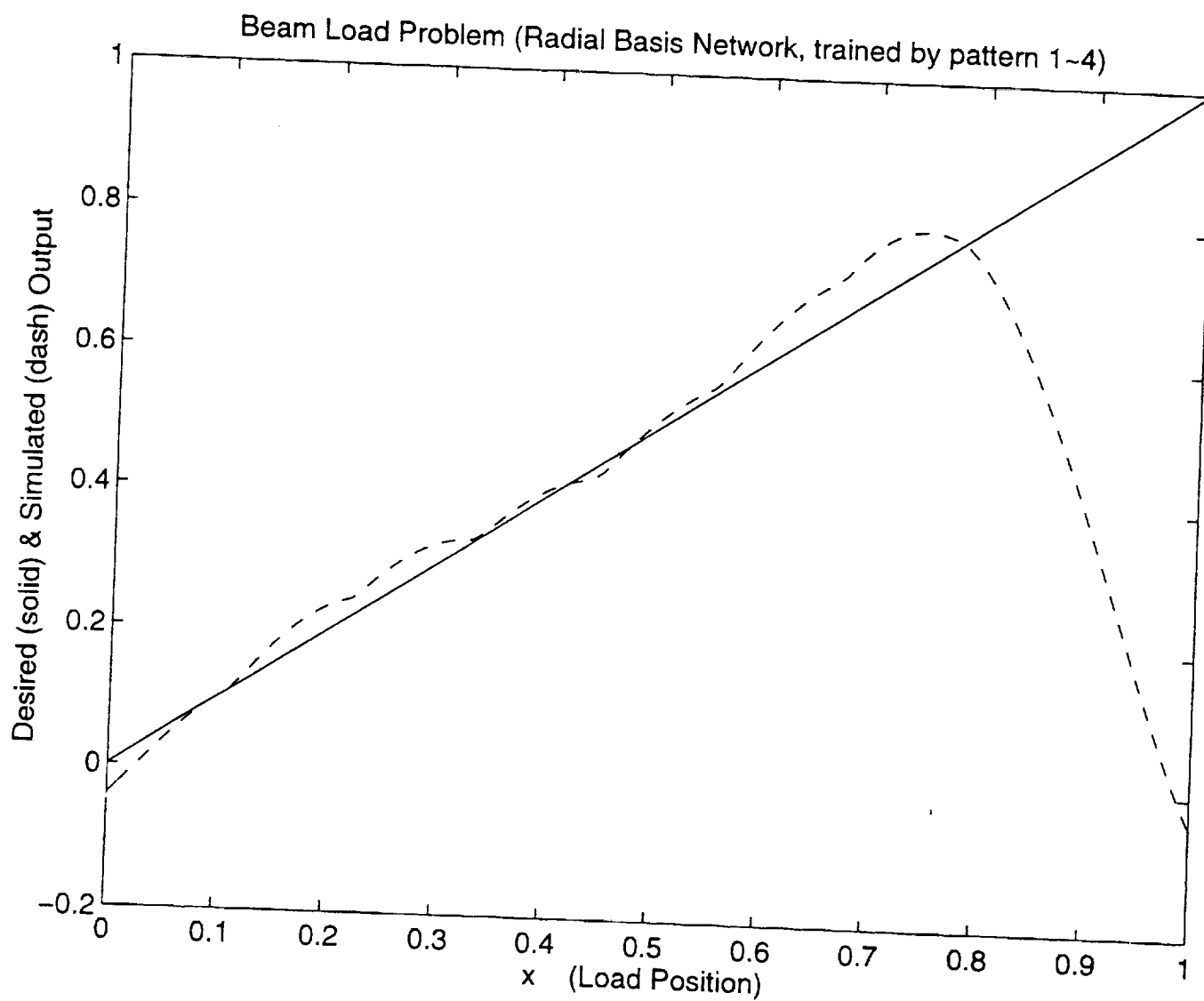


Fig. 16 5-6-6-1 NN mapping of a structural design problem:
(a) the present results

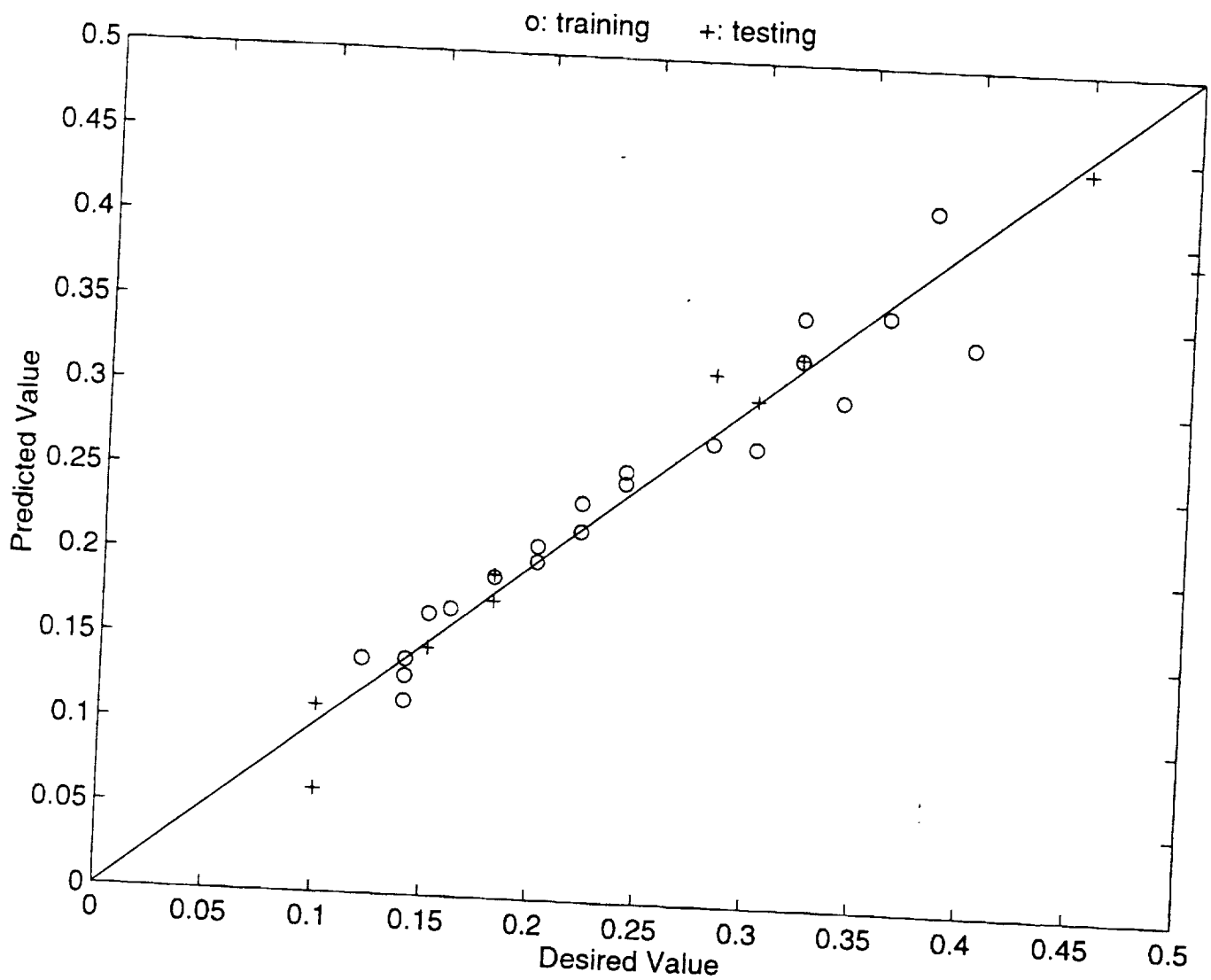


Fig. 16 5-6-6-1 NN mapping of a structural design problem:
(b) results from [8]

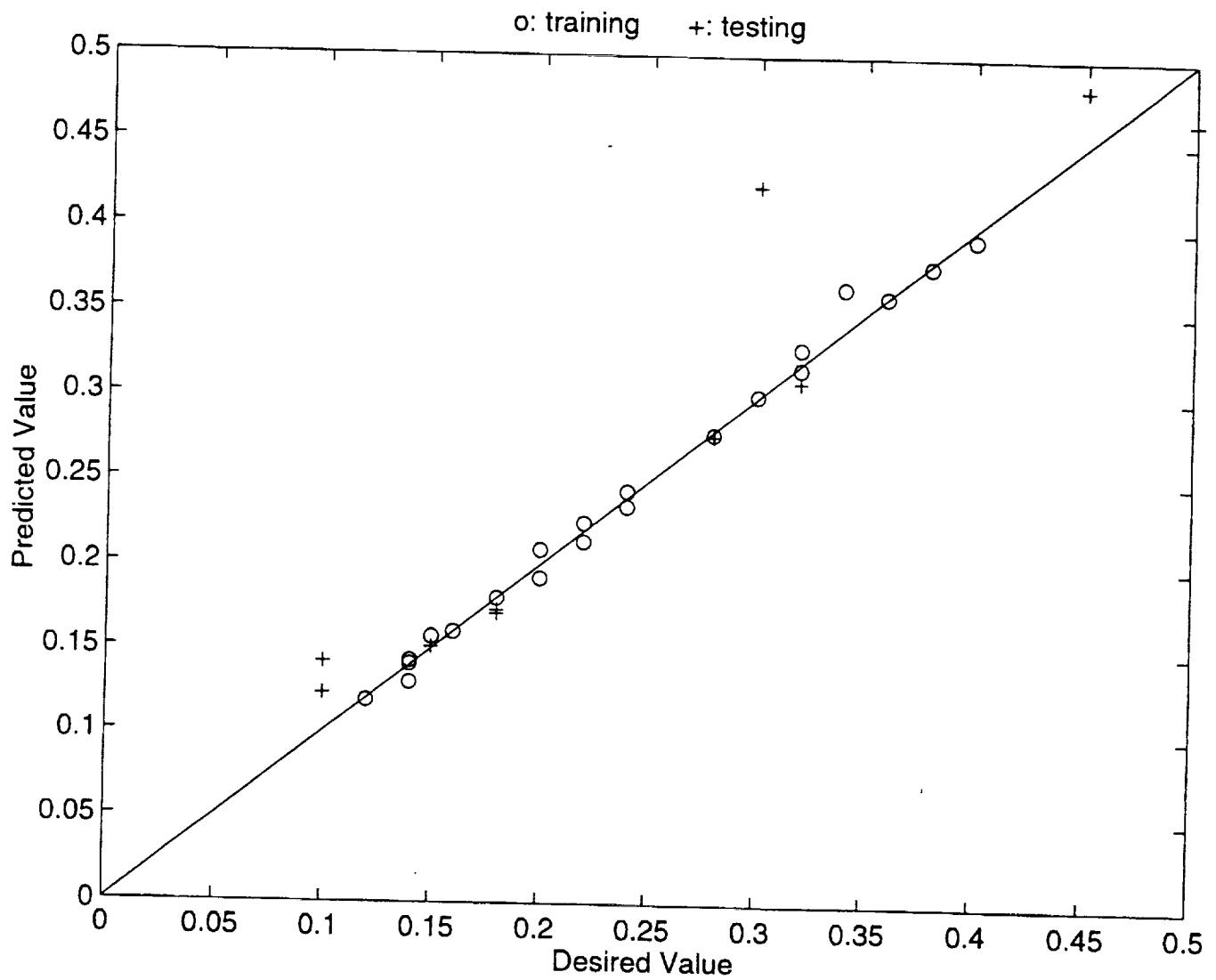


Fig. 17 1-9-1 NN mapping of a structural design problem:
(a) the present results

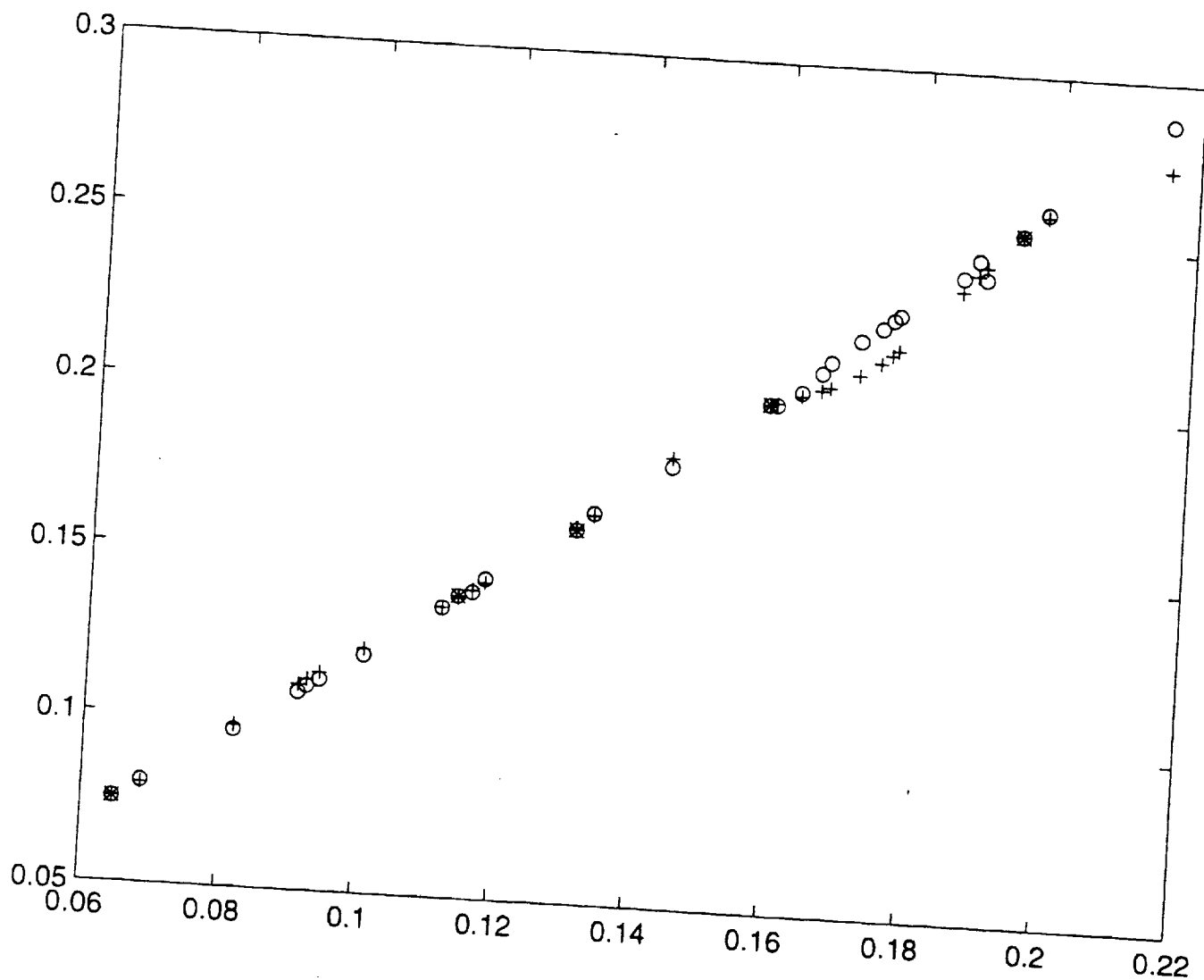


Fig. 17 1-9-1 NN mapping of a structural design problem:
(b) results from [9]

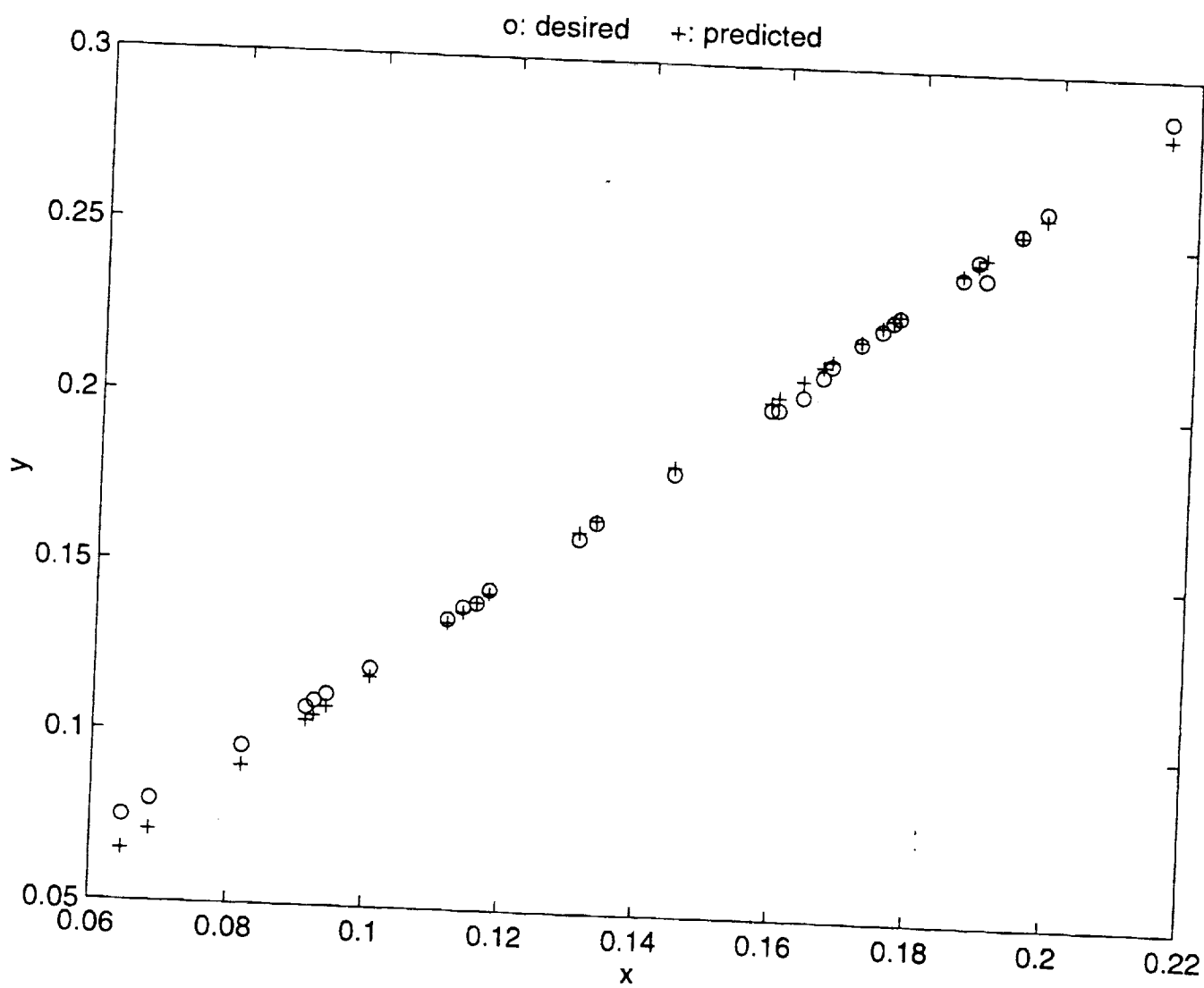


Fig. 18(a) Feed-forward NN modeling of a sine wave with noises (1-20-20-1)
(+: training data; solid line: network result; dashed line: true function)

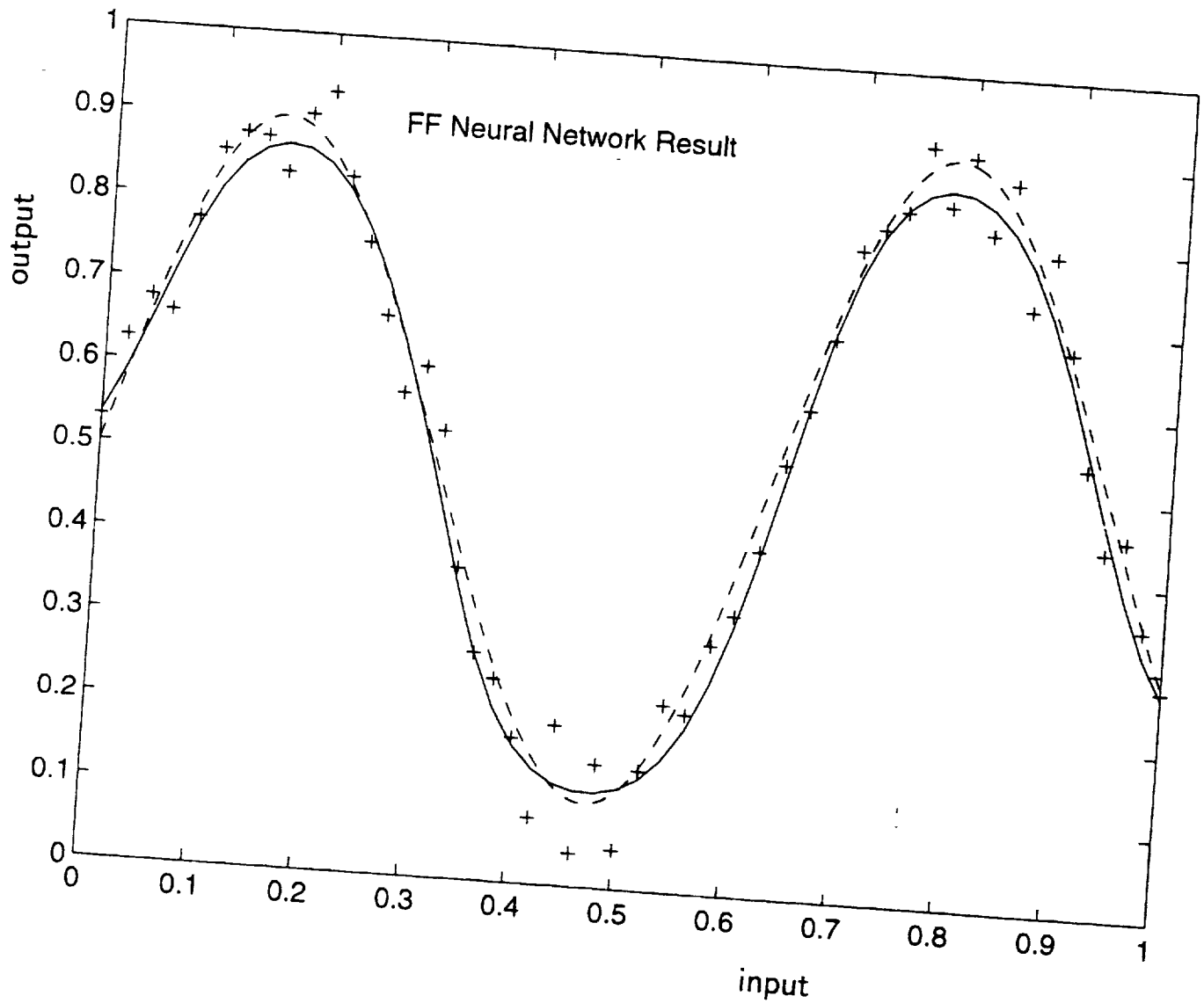


Fig. 18(b) Radial Basis Function NN modeling of a sine wave with noises
 (+: training data; solid line: network result; dashed line: true function)

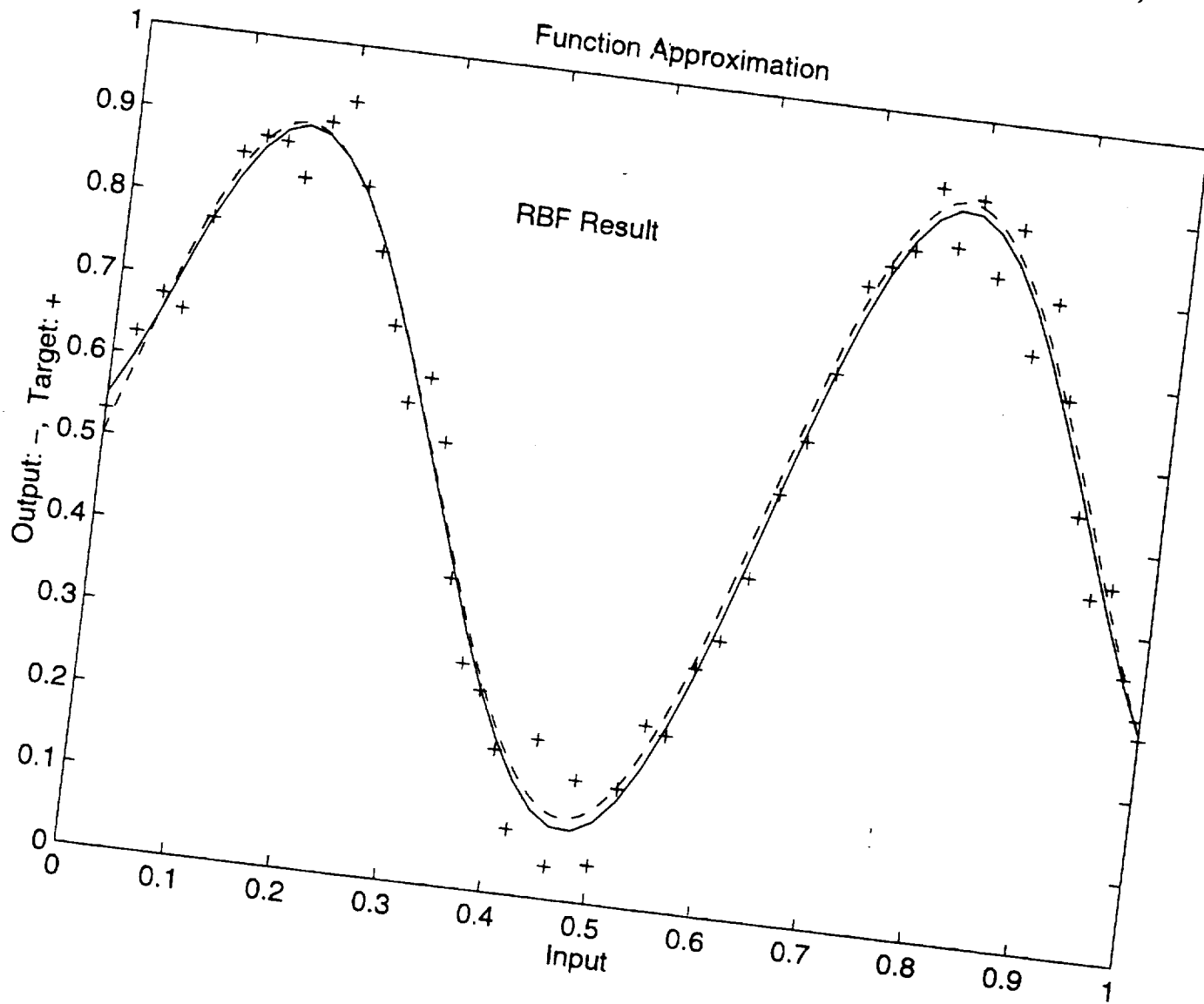


Fig. 19(a) Feed-forward NN modeling of a sine wave with noises (1-20-20-1)
(+: training data; solid line: network result; dashed line: true function)

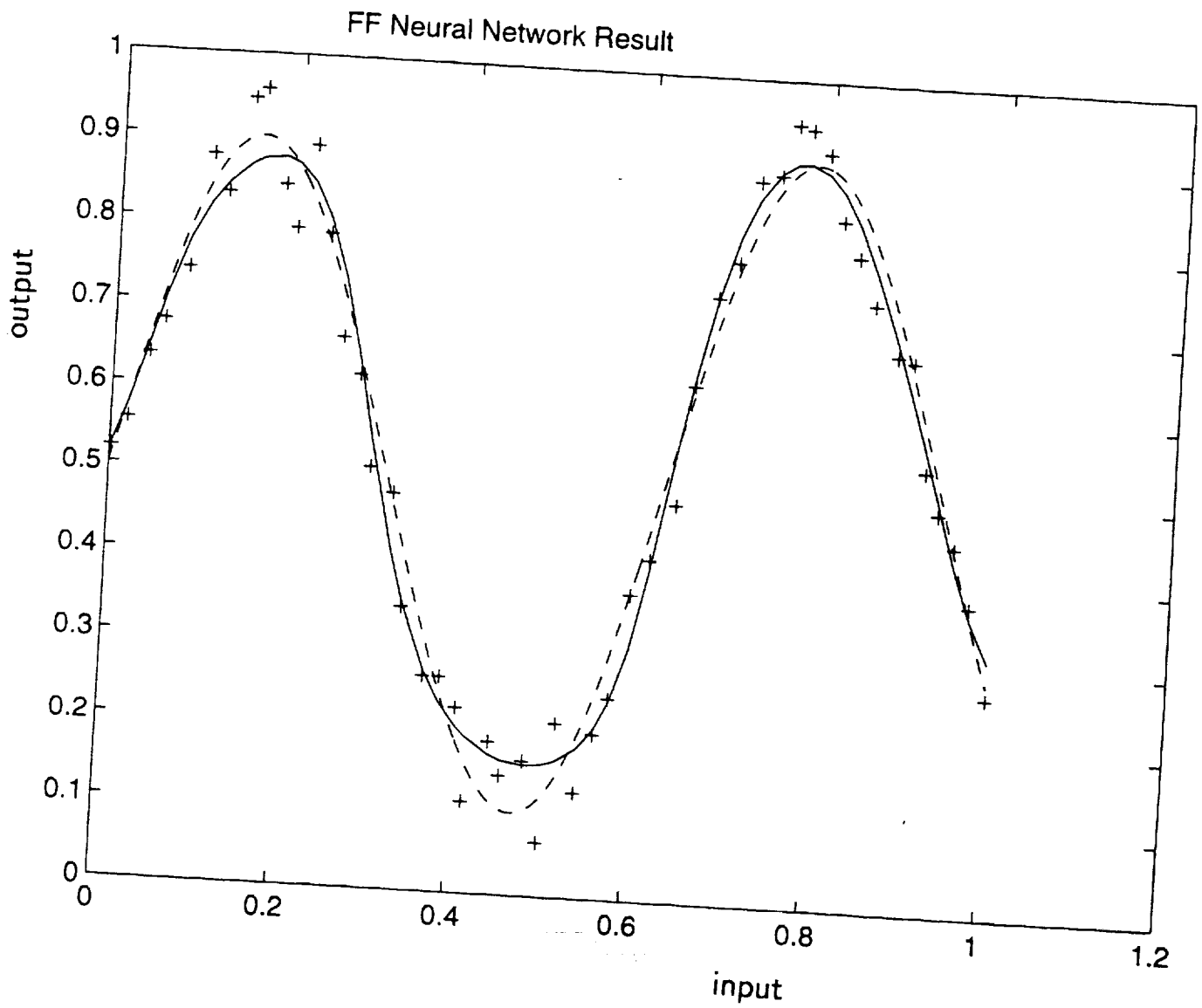


Fig. 19(b) Radial Basis Function NN modeling of a sine wave with noises
(+: training data; solid line: network result; dashed line: true function)

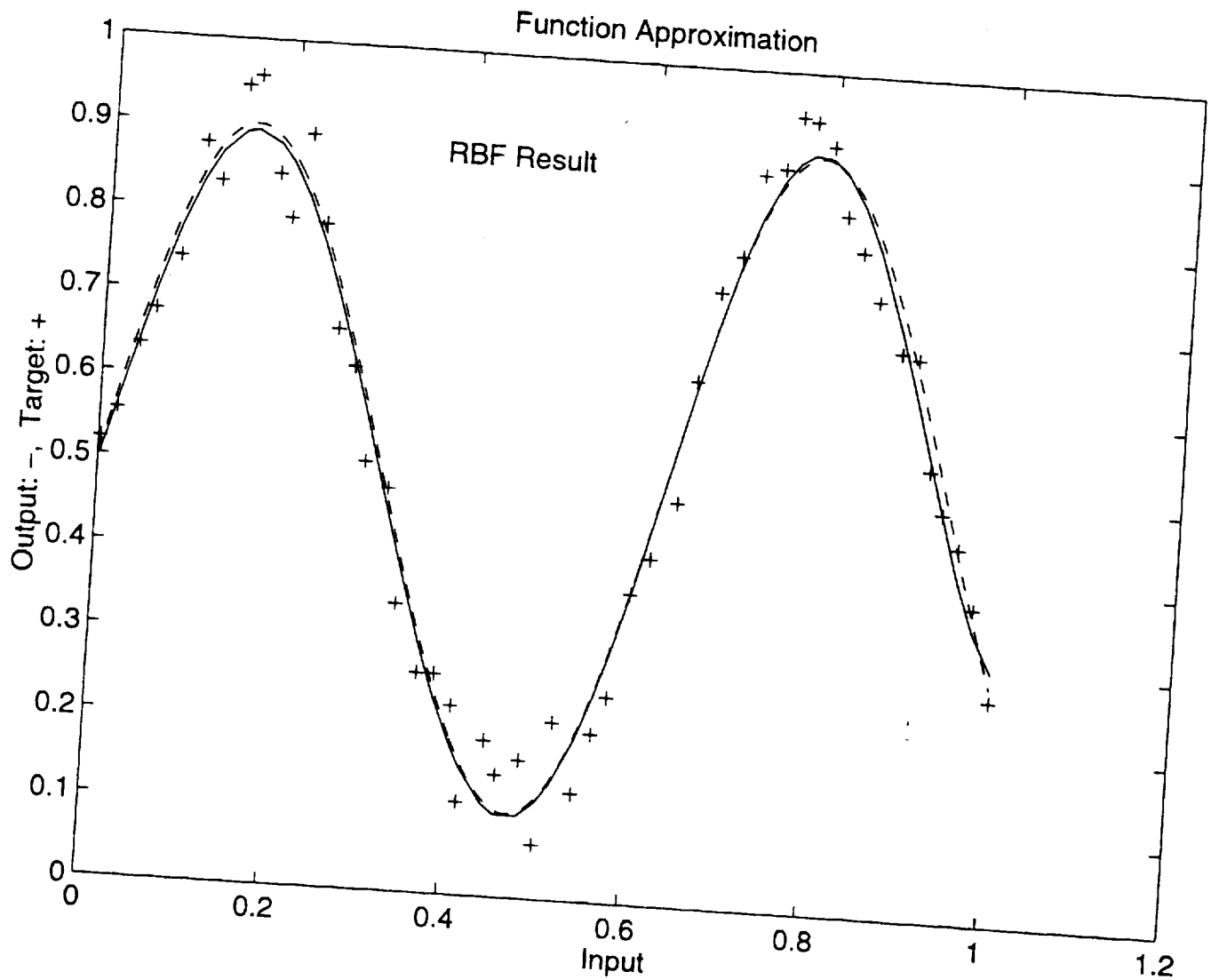


Fig. 19(c) Summation of network results from Fig. 23(a) and (b)

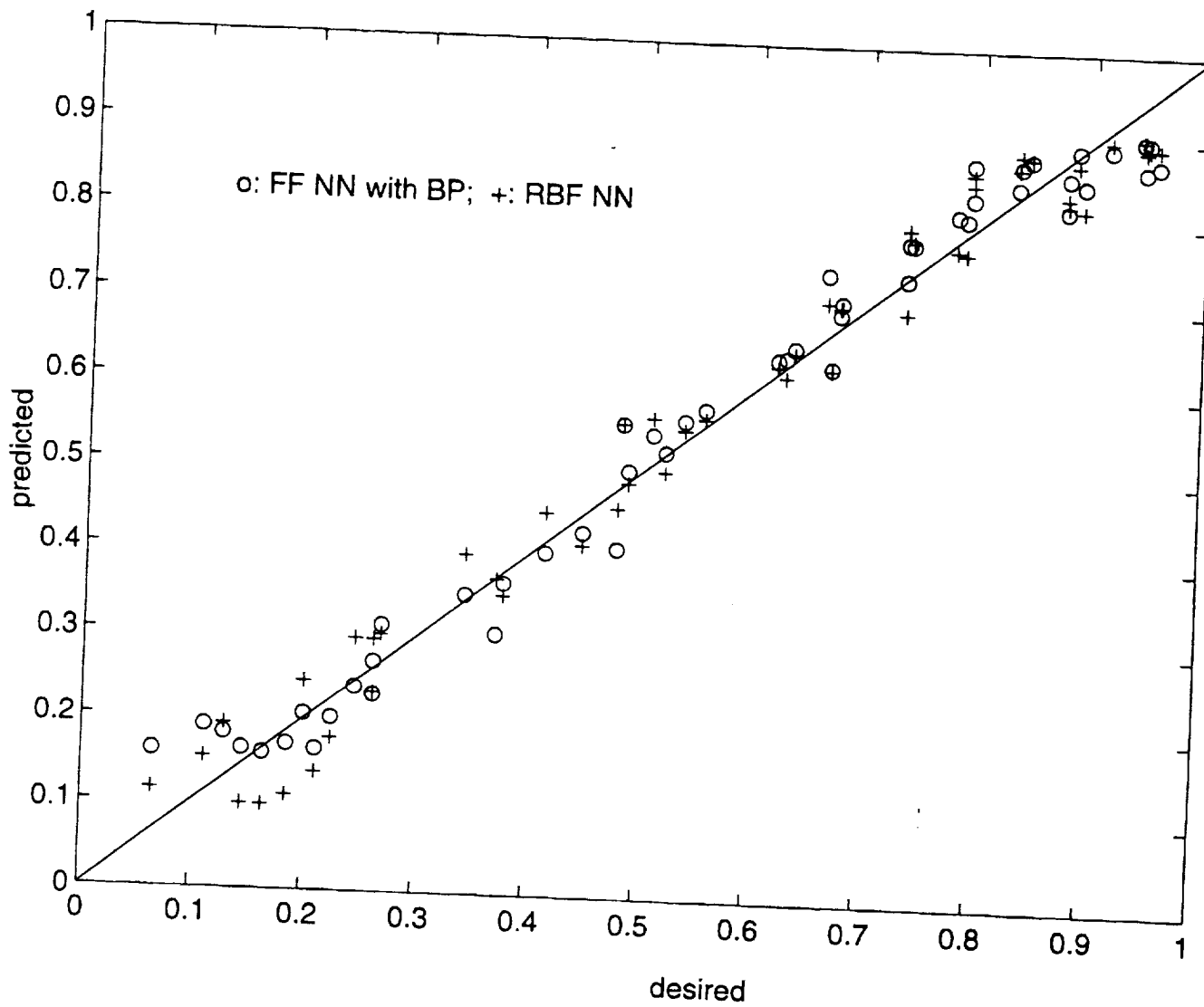


Fig. 19(d) Feed-forward NN modeling of a sine wave with noises (1-49-1)
 (+: training data; solid line: network result; dashed line: true function)

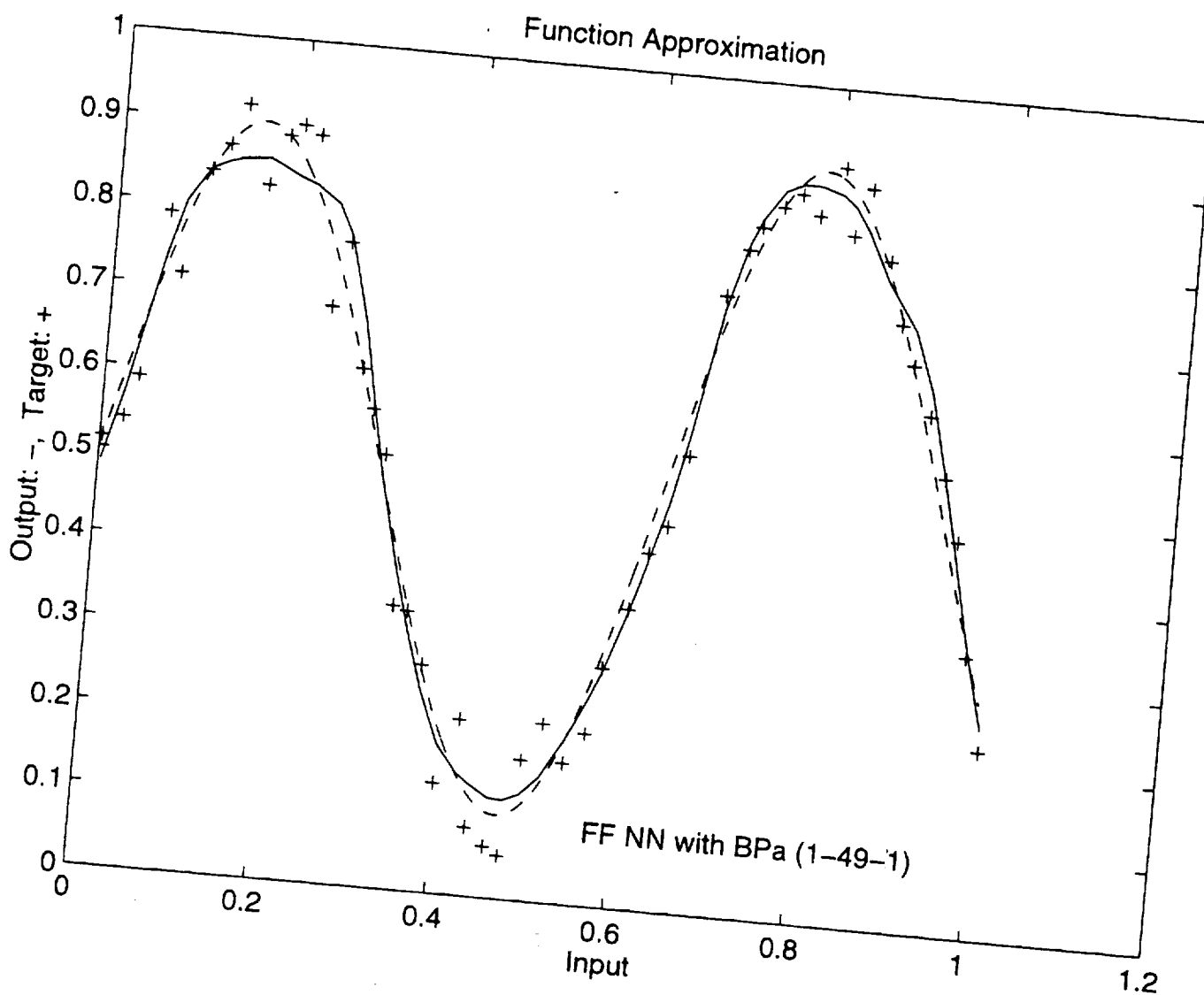


Fig. 20 Feed-forward NN with Back-propagation, Active-learning and
Transfer Function Combination *Formulation I*
(a) One hidden layer (1-49-1)

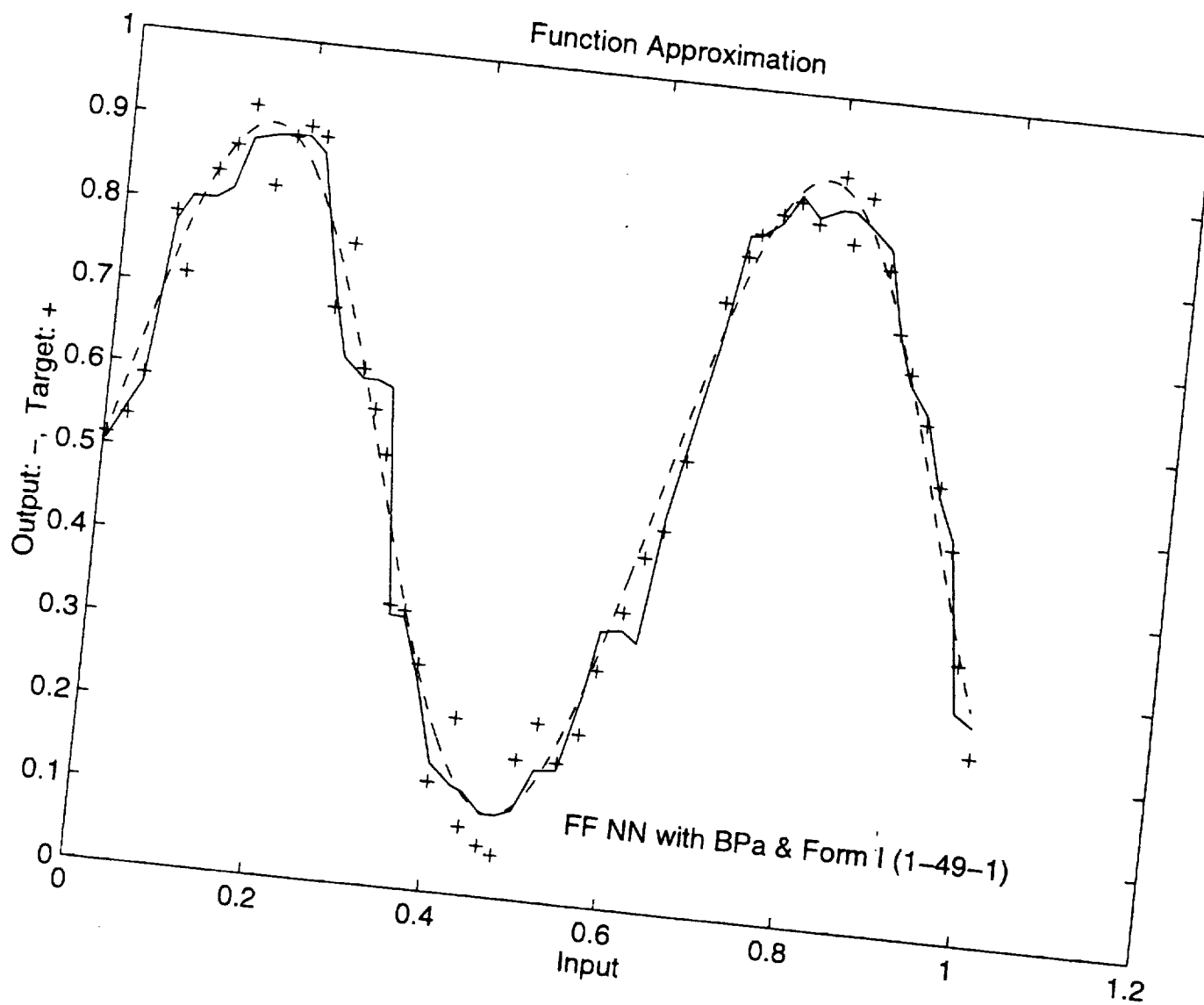


Fig. 20 Feed-forward NN with Back-propagation, Active-learning and
Transfer Function Combination *Formulation I*
(b) Two hidden layers (1-20-20-1)

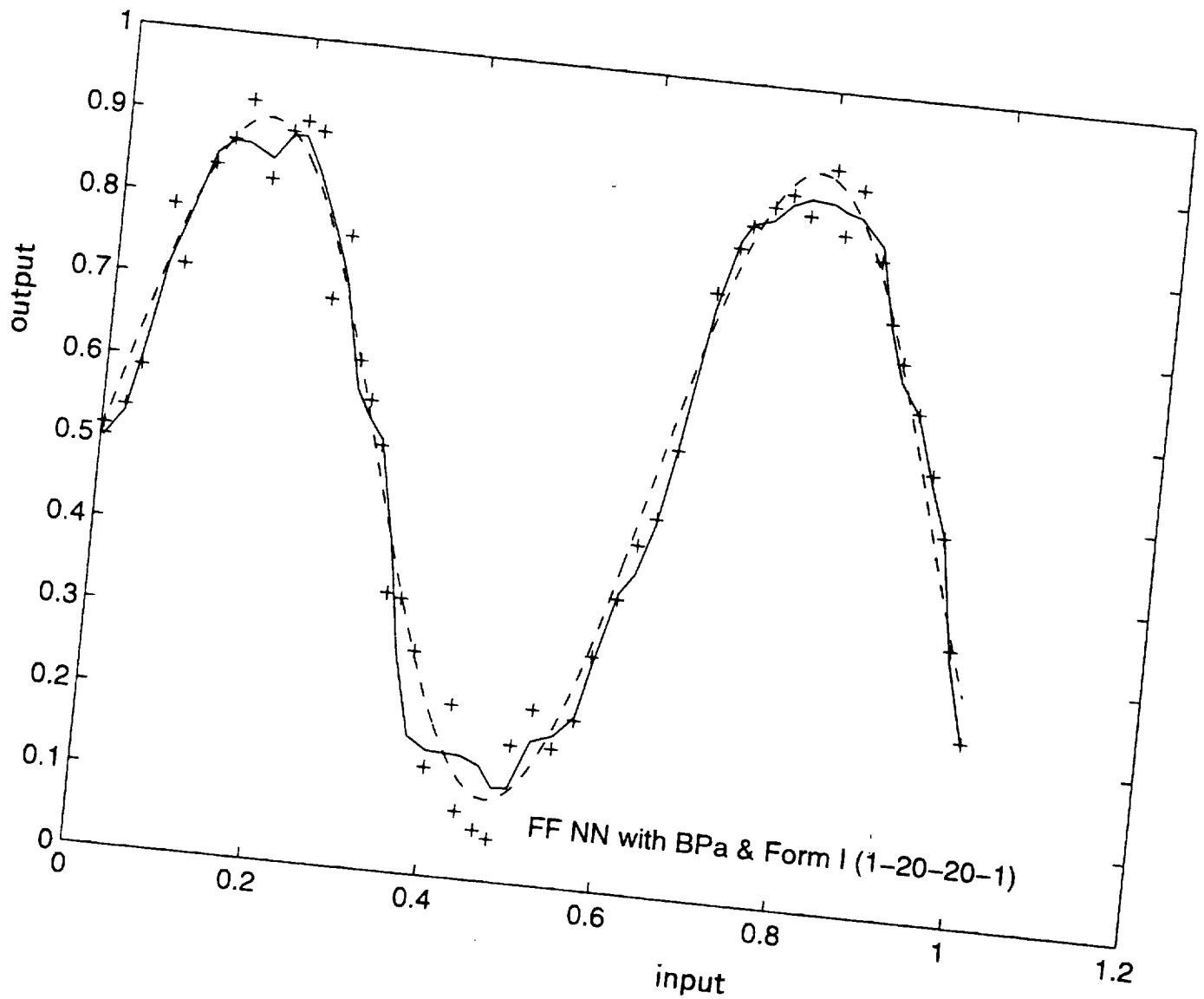


Fig. 21(a) Solving the beam ultimate moment capacity problem (5.4.2.1) with different schemes and results:
5-50-50-1 NN, very small error goal, epoch $\sim 10^4$

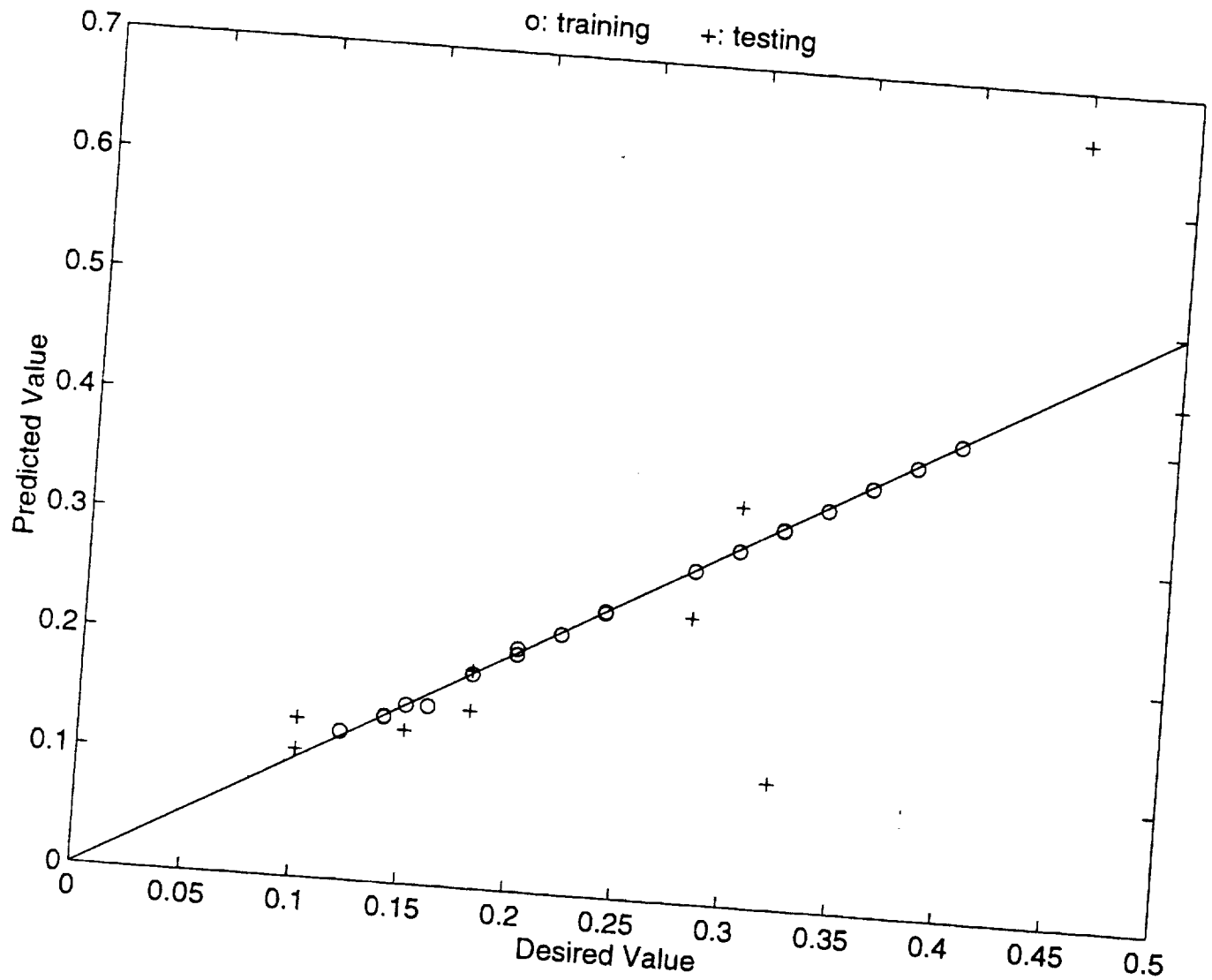


Fig. 21(b) Solving the problem in 5.4.2.1 with different schemes and results: 5-10-10-1 NN. quite large error goal, epoch=2000

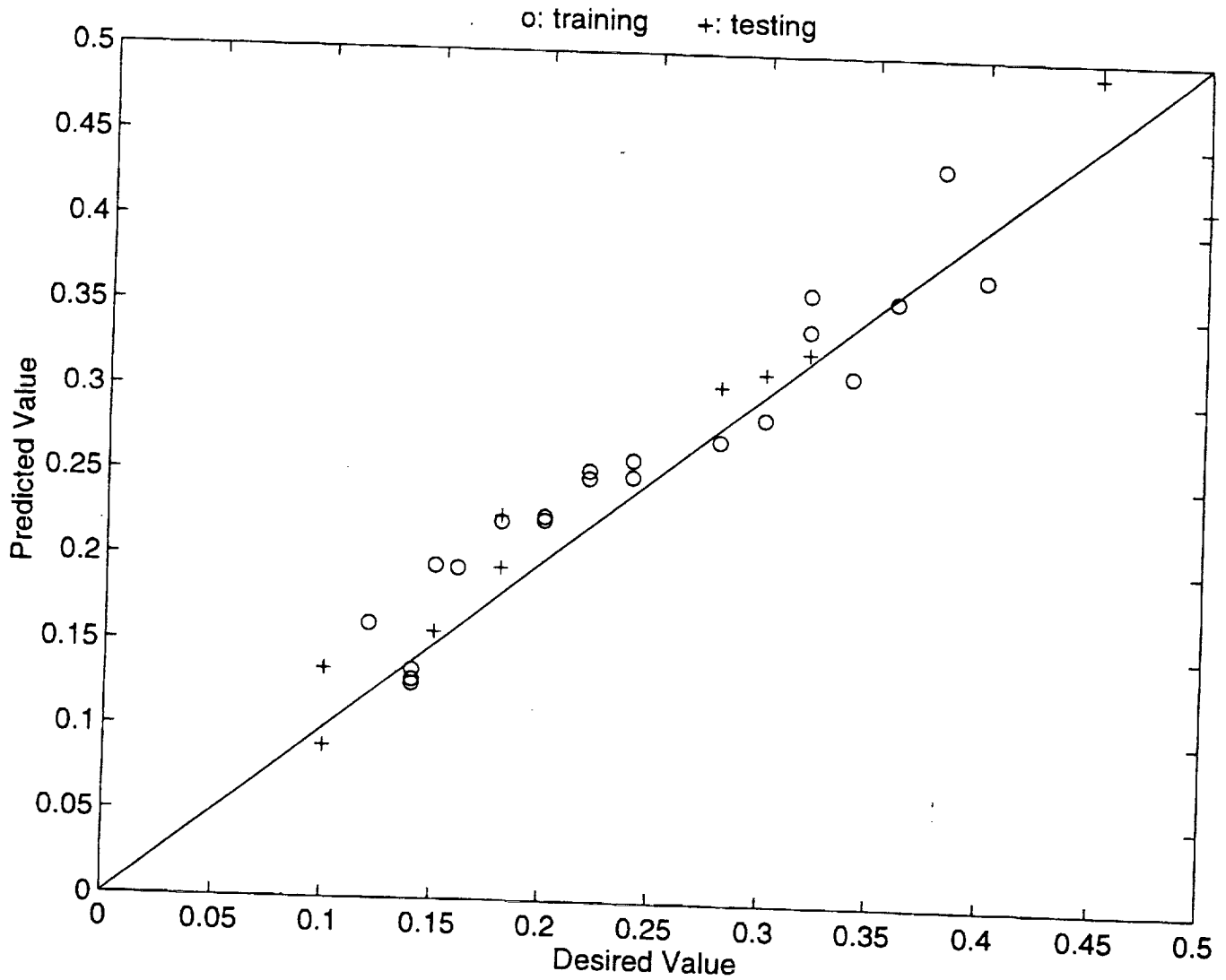


Fig. 21(c) Solving the problem in 5.4.2.1 with different schemes and results: 5-20-20-1 NN, quite large error goal, epoch=2000

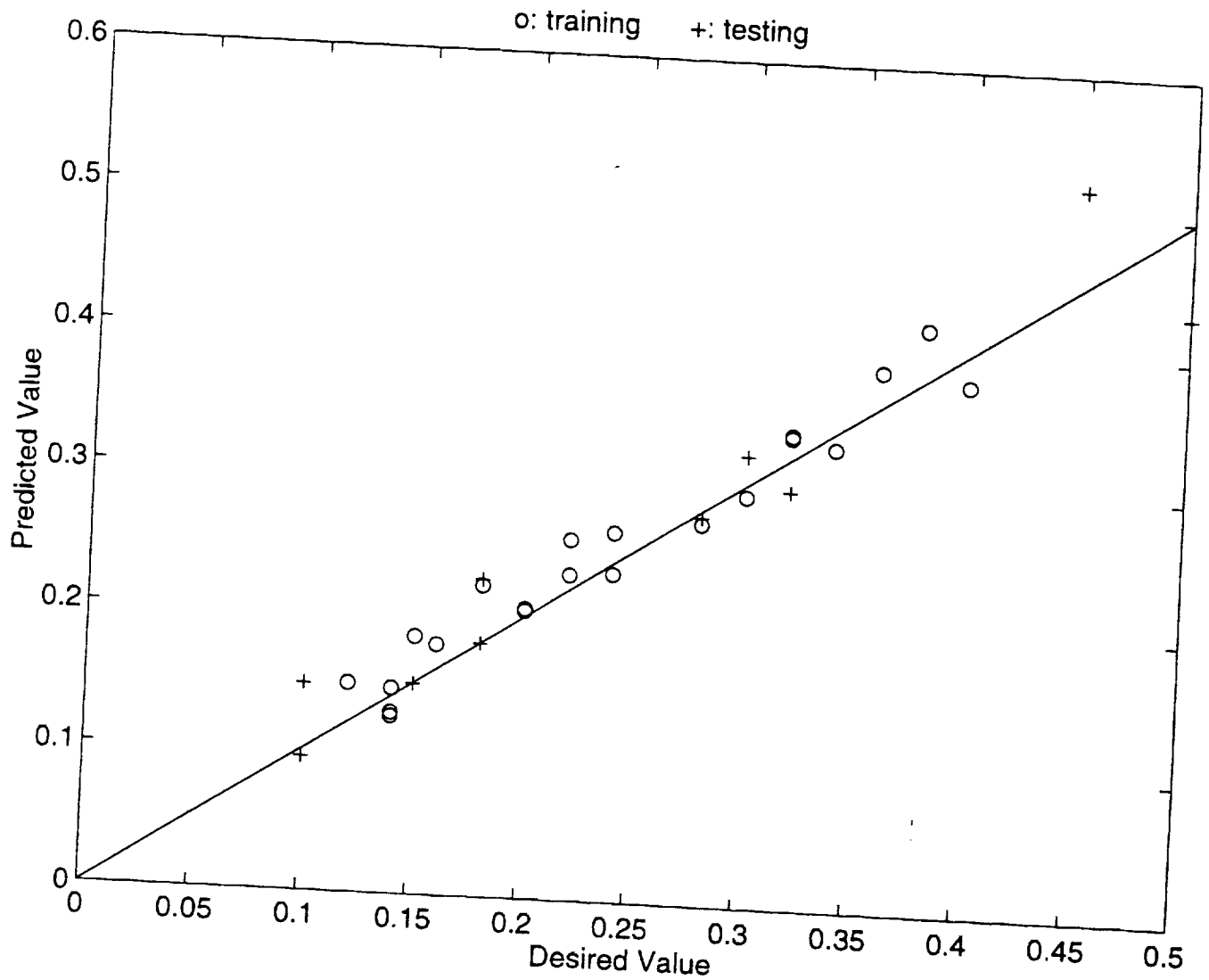
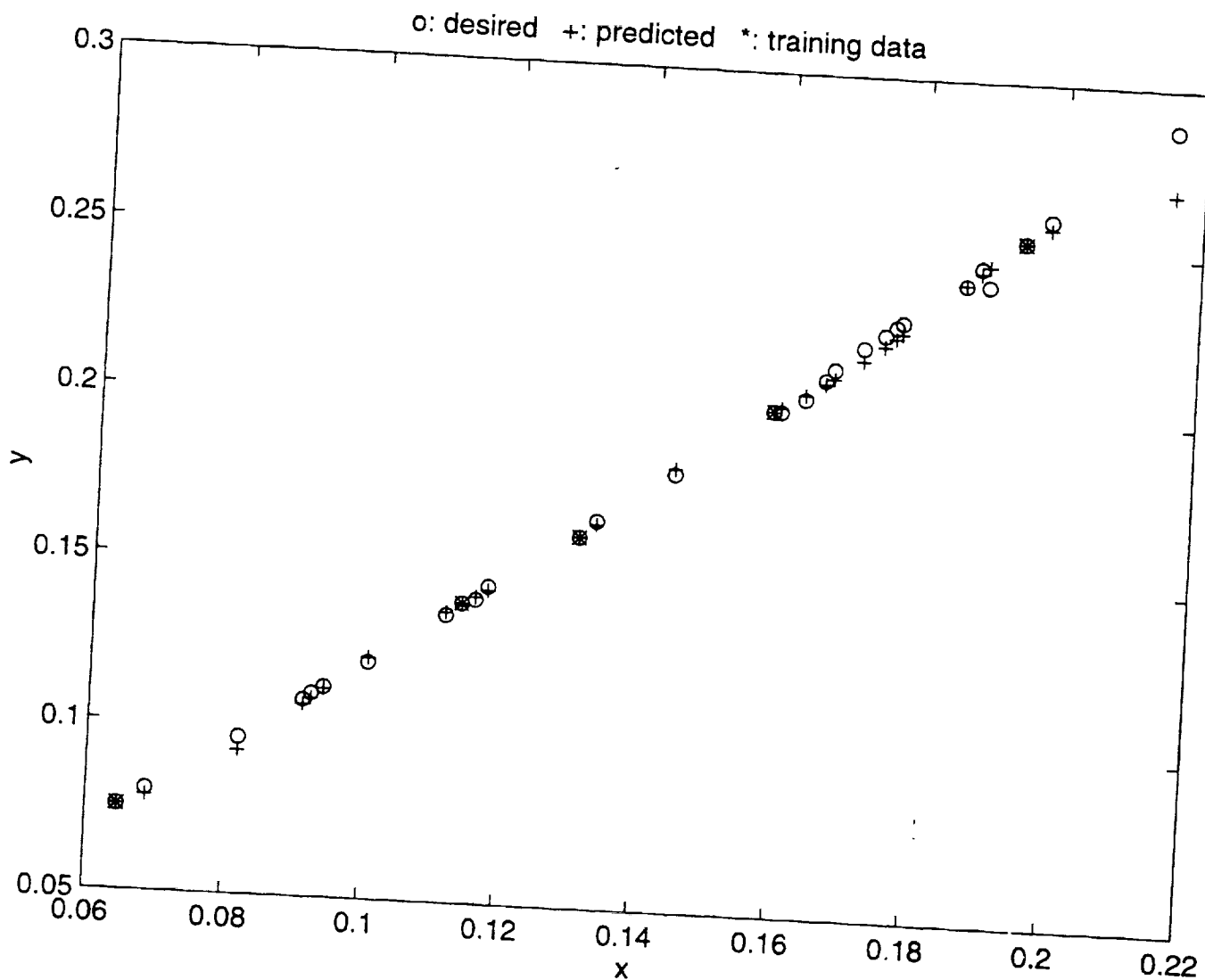


Fig. 22(a) Normalization of input and output variables: the problem in 5.4.2.2. 1-9-1, 4*p. 3*t. epoch=468



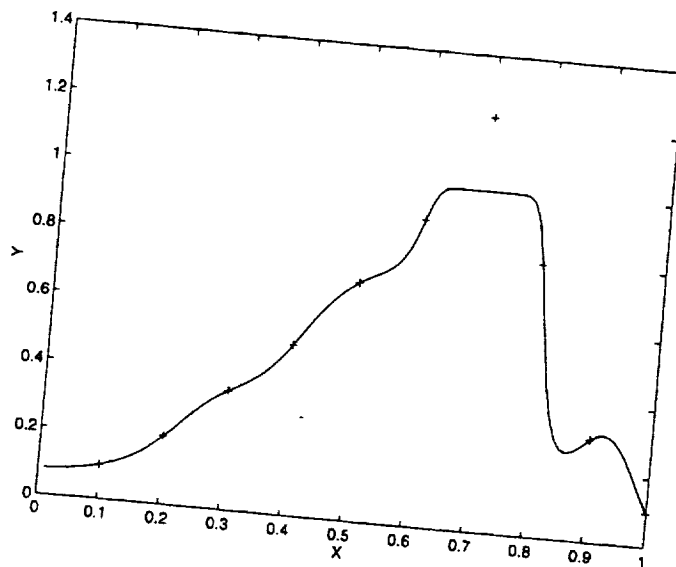


Fig. 22 (b) Normalization of input and output variables: 'logsig', 1-10-1, training data $t(.7) = 1.22 > 1$, symbols: desired values, line: simulation

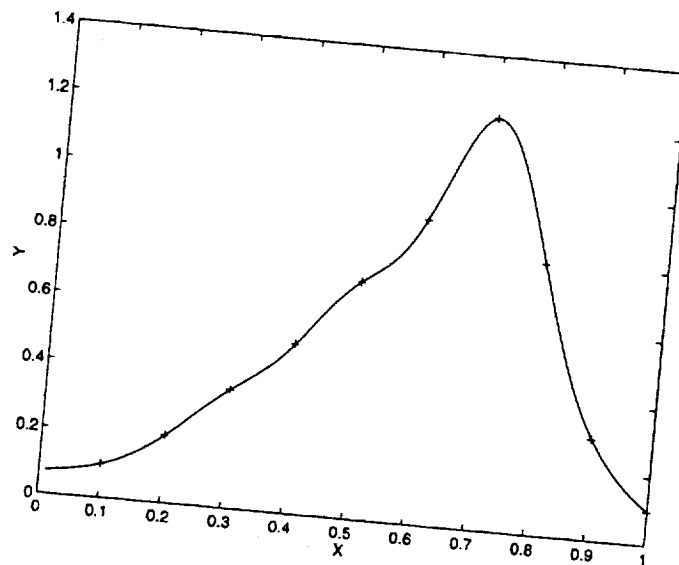


Fig. 22 (c) Normalization of input and output variables: 'purelin'

Fig. 23 Solving the problem in 5.4.2.2 by Formula I: 1-9-1
 NN. error goal= 10^{-6} , epoch=8851

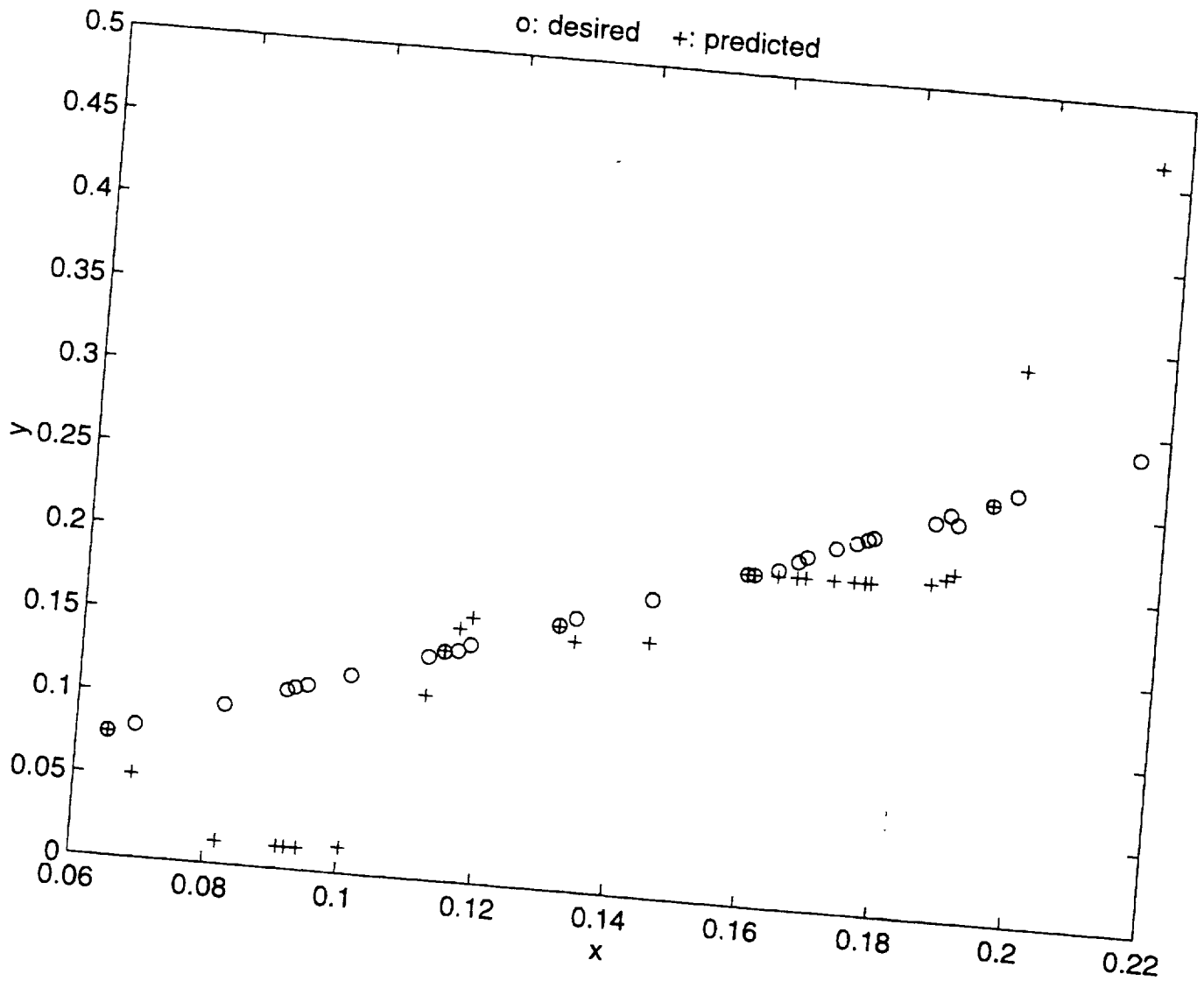


Fig. 24(a) Solving again the problem in 5.4.2.2: 1-5-1 NN,
error goal= 10^{-6} , epoch=15117

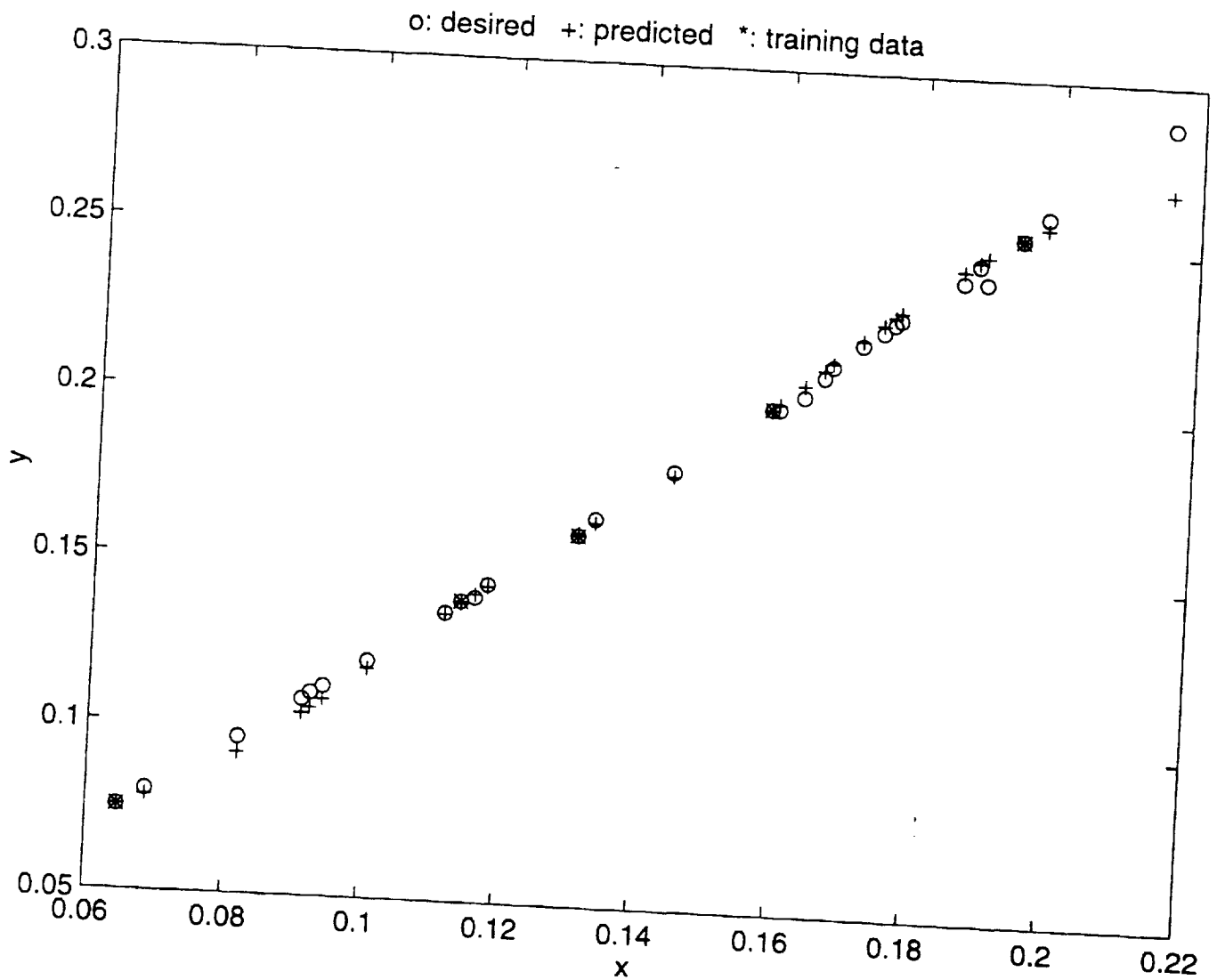


Fig. 24(b) Solving again the problem in 5.4.2.2: 1-6-1 NN,
error goal= 10^{-6} . epoch=2155

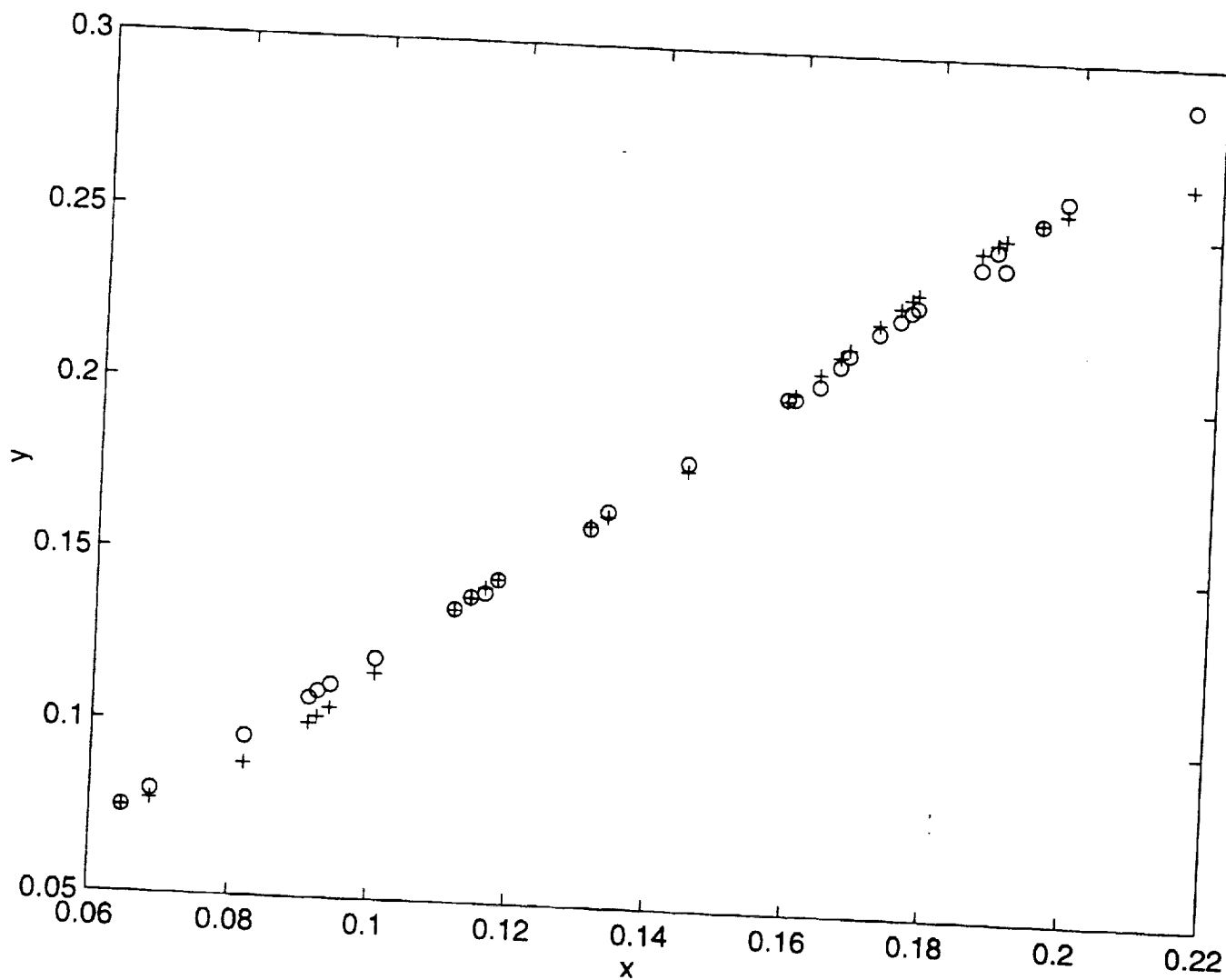


Fig. 24(c) Solving again the problem in 5.4.2.2: 1-9-1 NN,
error goal= 10^{-6} , epoch=5326

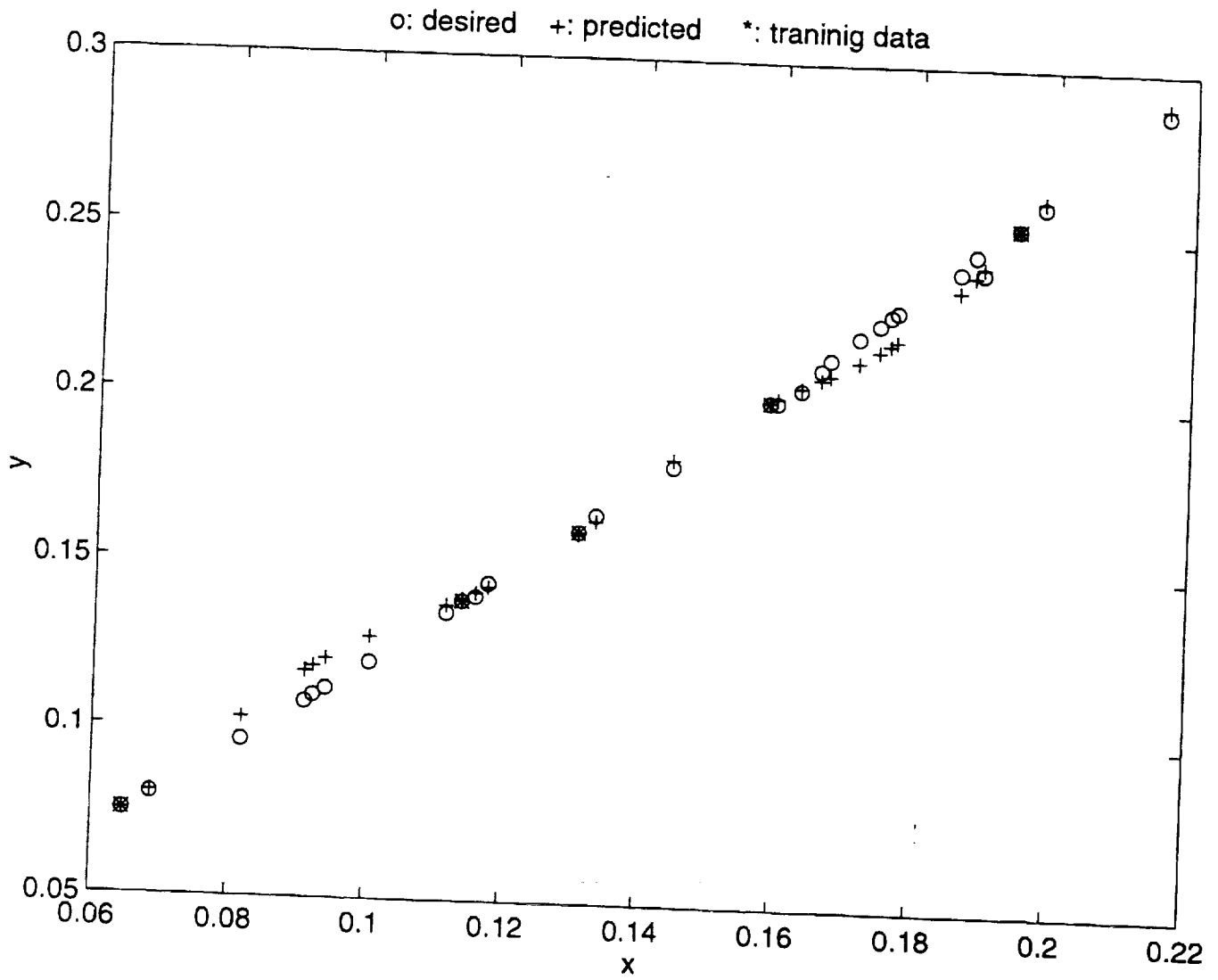


Fig. 24(d) Solving again the problem in 5.4.2.2: 1-11-1 NN,
error goal= 10^{-6} , epoch=2401

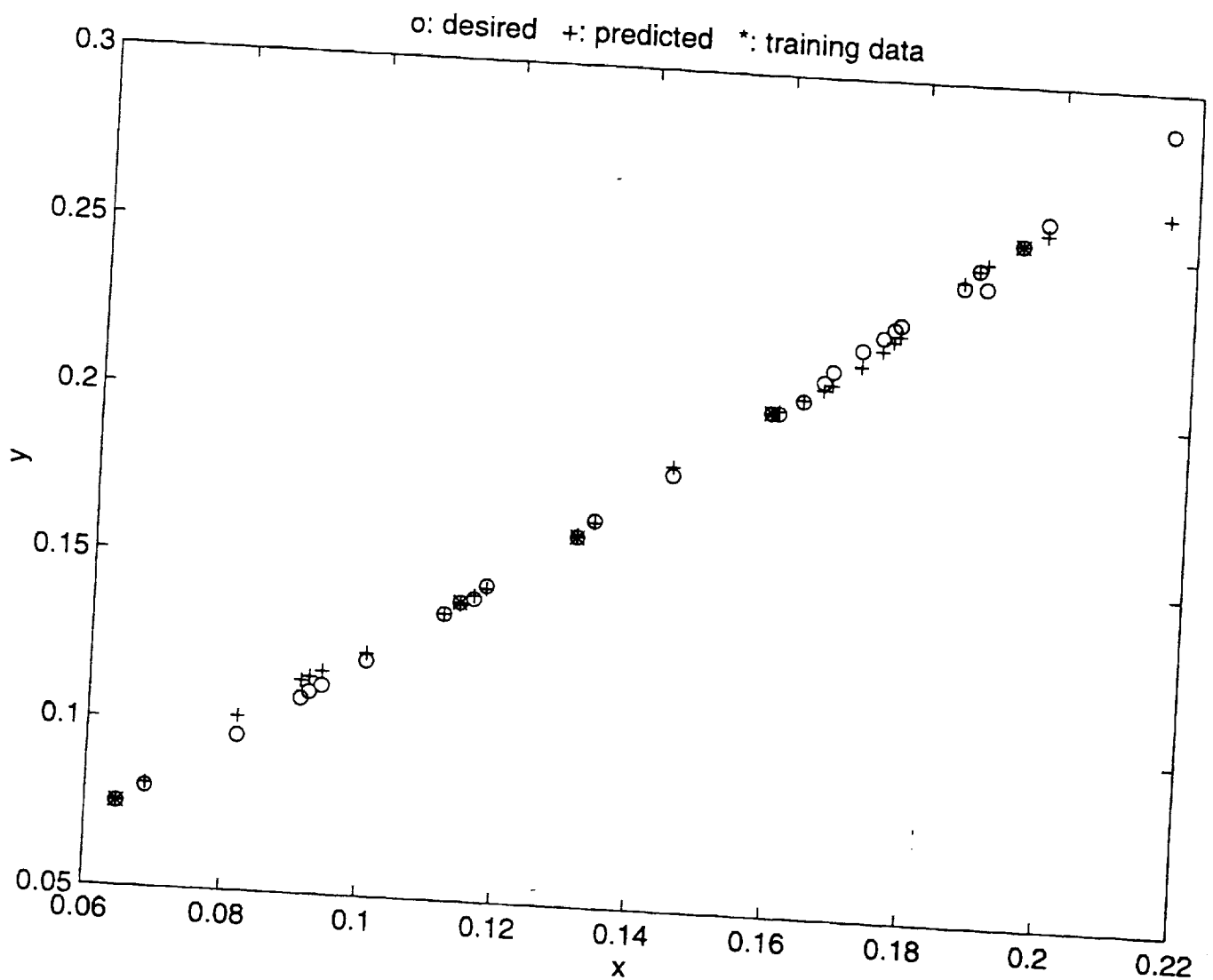


Fig. 24(e) Solving again the problem in 5.4.2.2: 1-14-1 NN,
error goal= 10^{-6} . epoch=427

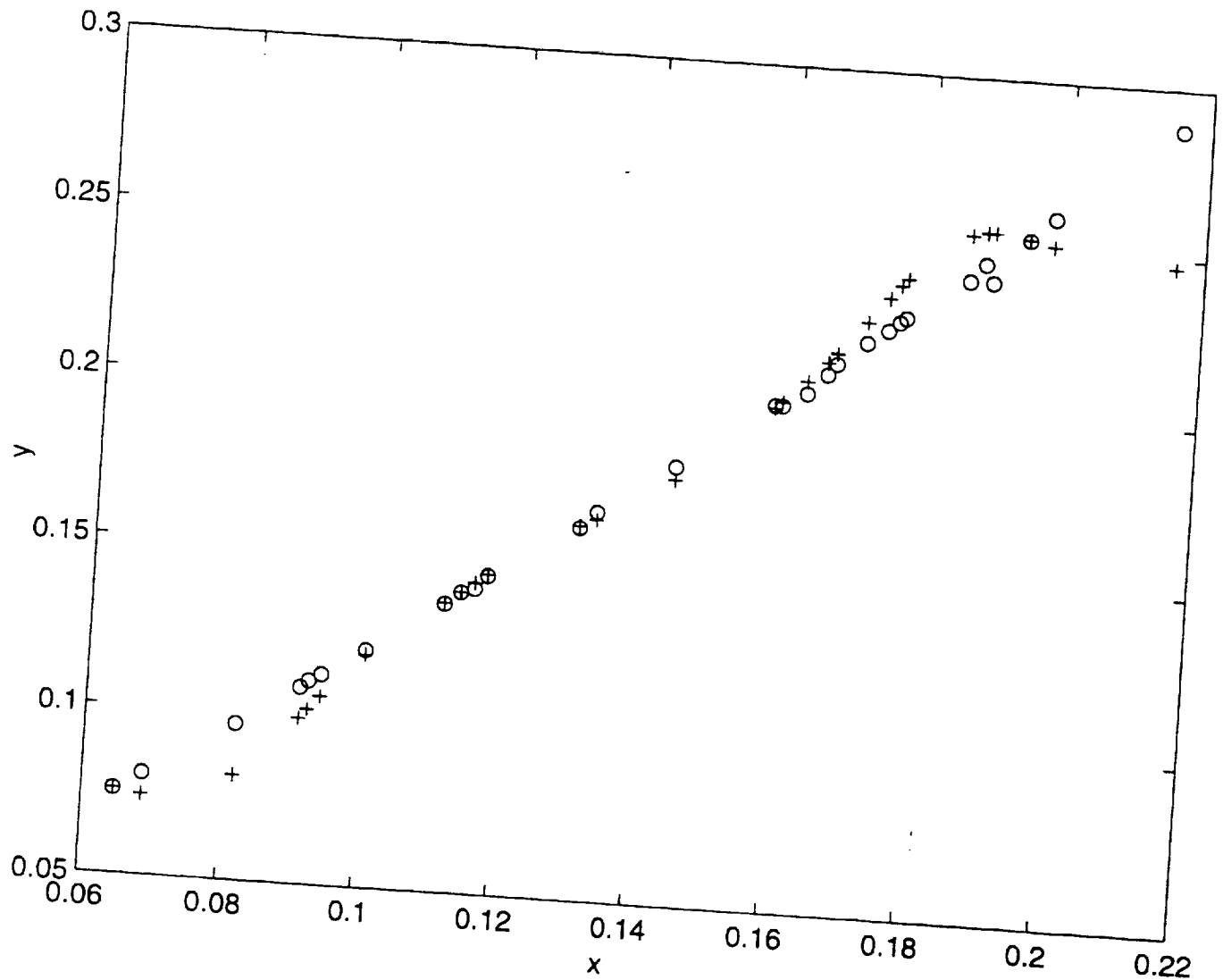


Fig. 24(f) Solving again the problem in 5.4.2.2: 1-19-1 NN,
error goal= 10^{-6} , epoch=971

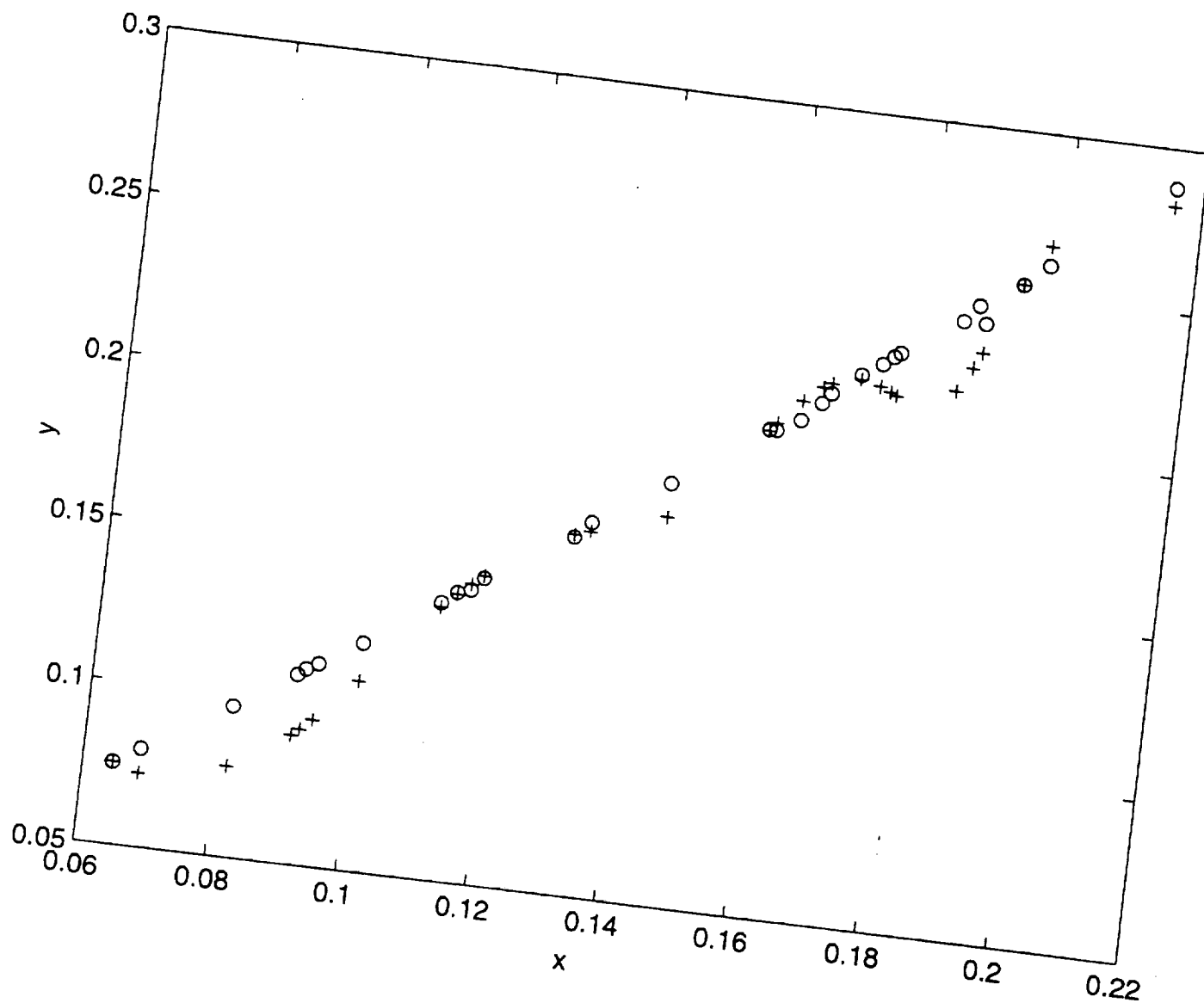


Fig. 24(g) Solving again the problem in 5.4.2.2: 1-9-9-1 NN,
error goal= 10^{-6} , epoch=5332

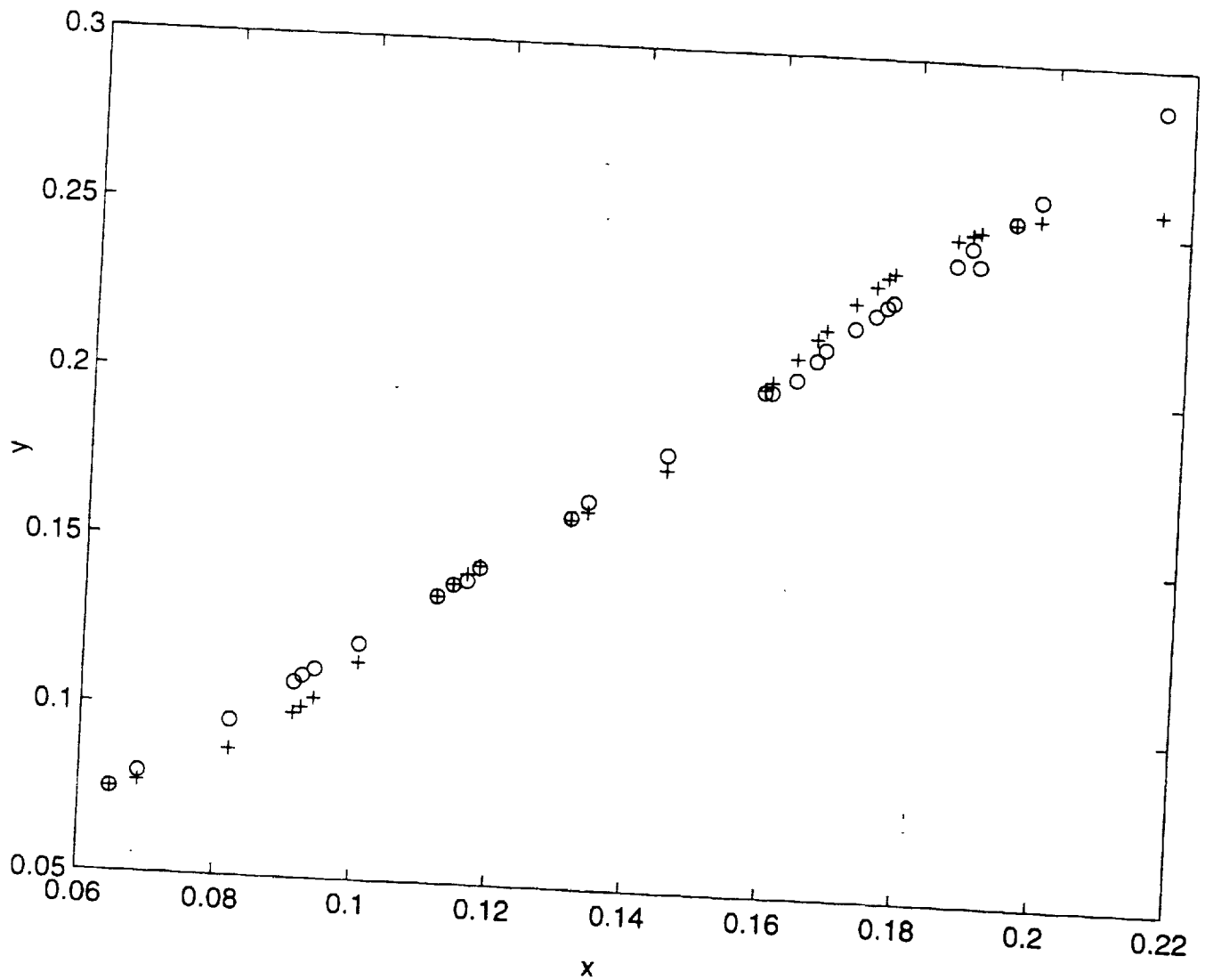


Fig. 24(h) Solving again the problem in 5.4.2.2: 1-19-19-1
NN. error goal= 10^{-6} . epoch=3021

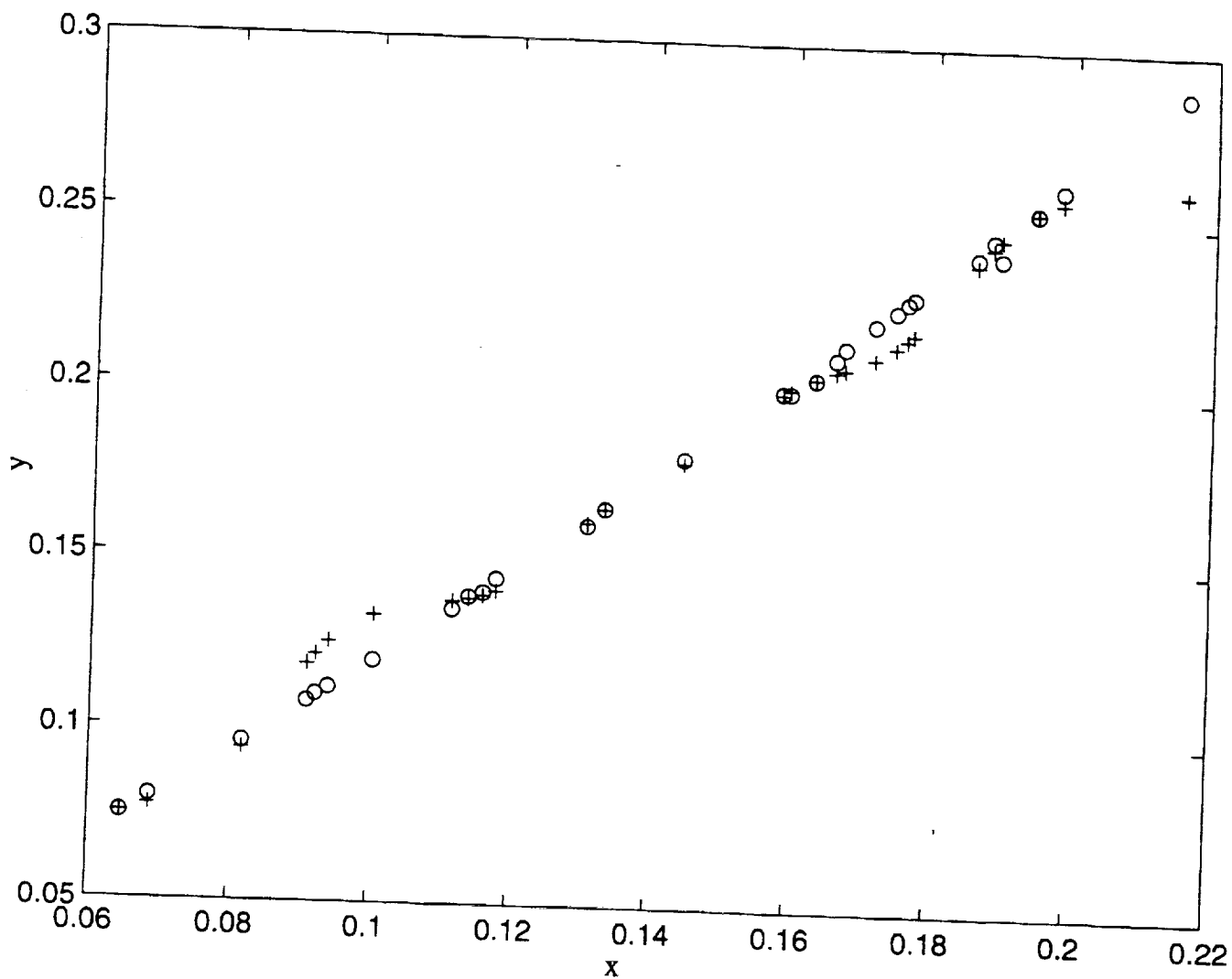


Fig. 25(a) Training History for the Toy Problem by a (1-20-1) NN:
Combined Errors

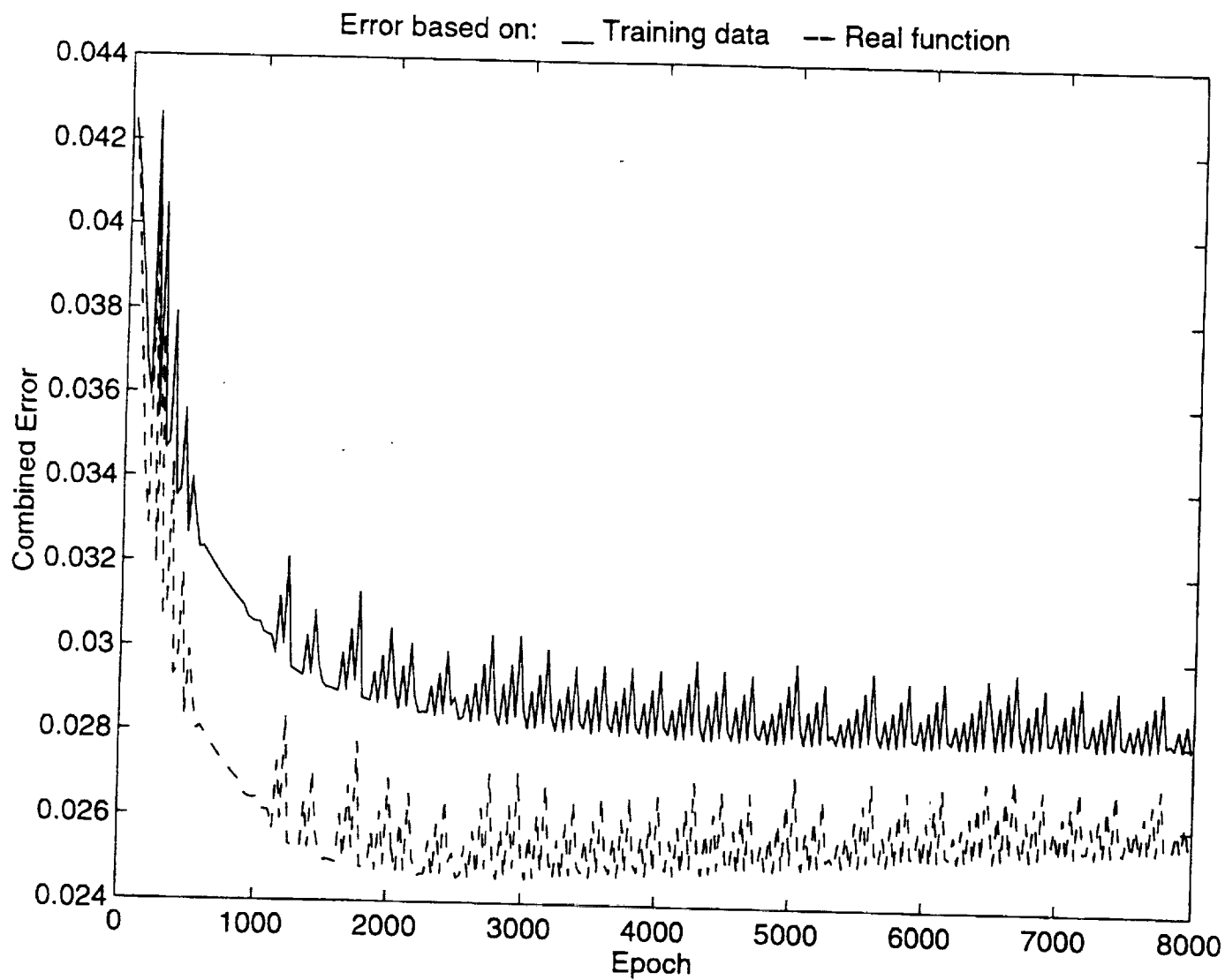


Fig. 25(b) Training History for the Toy Problem by a (1-20-1) NN:
Ratios of Testing Error over Training Error

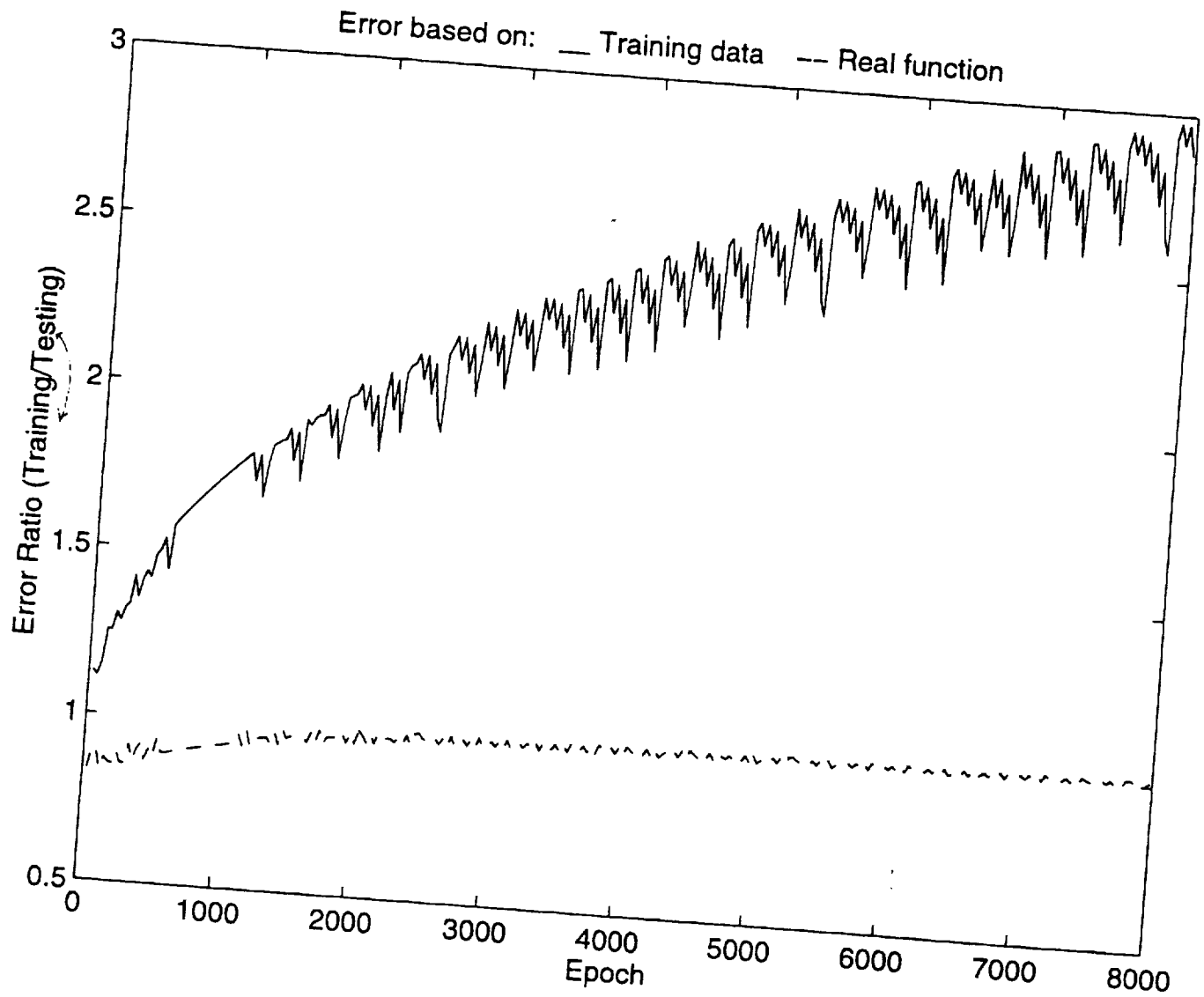


Fig. 25(c) Results for the Toy Problem by a (1-20-1) NN:
Comparison of the Prediction and the Target Values

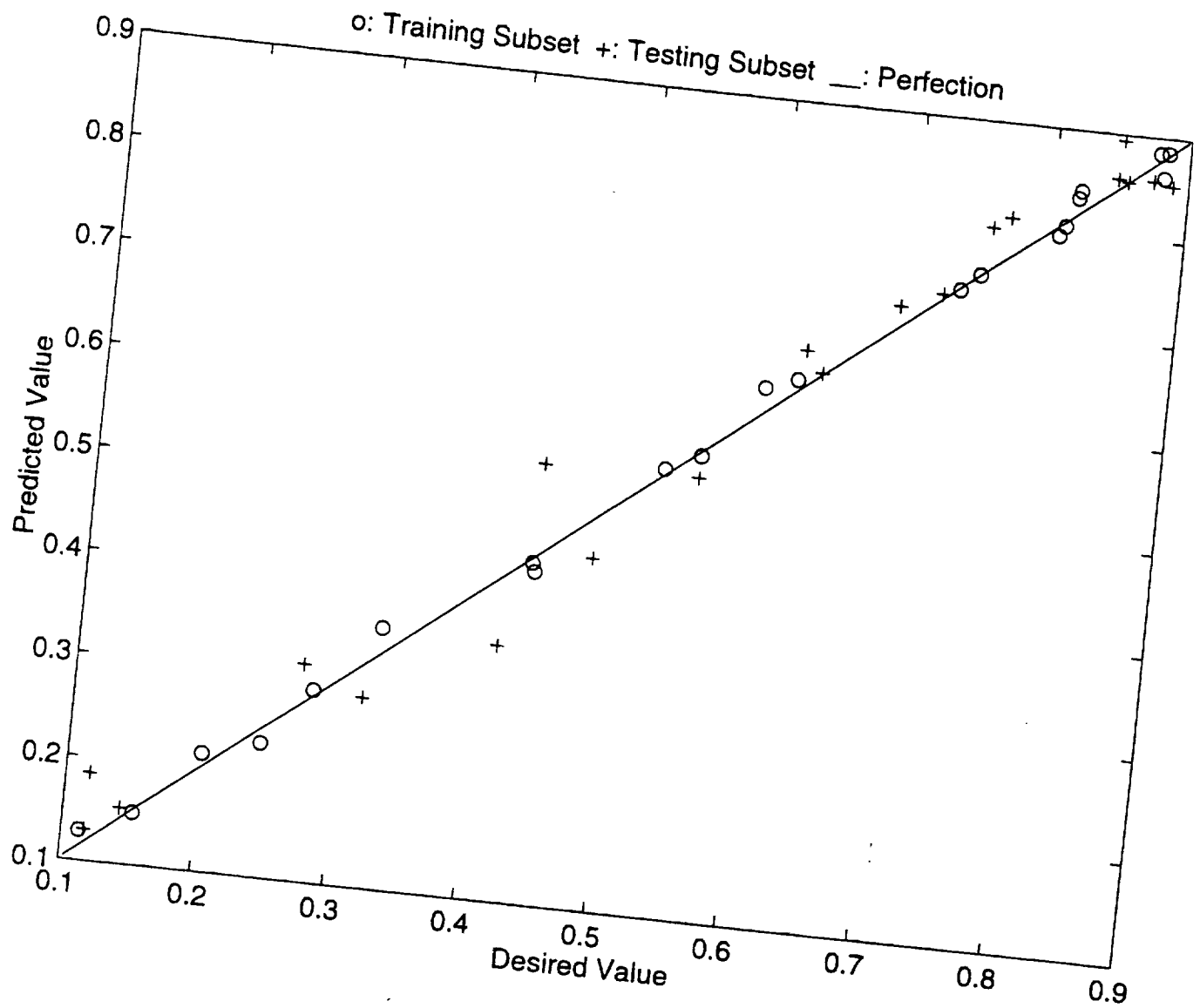


Fig. 25(d) Results for the Toy Problem by a (1-20-1) NN: Comparison of the Prediction, Training and Testing Subsets, and the Real Function

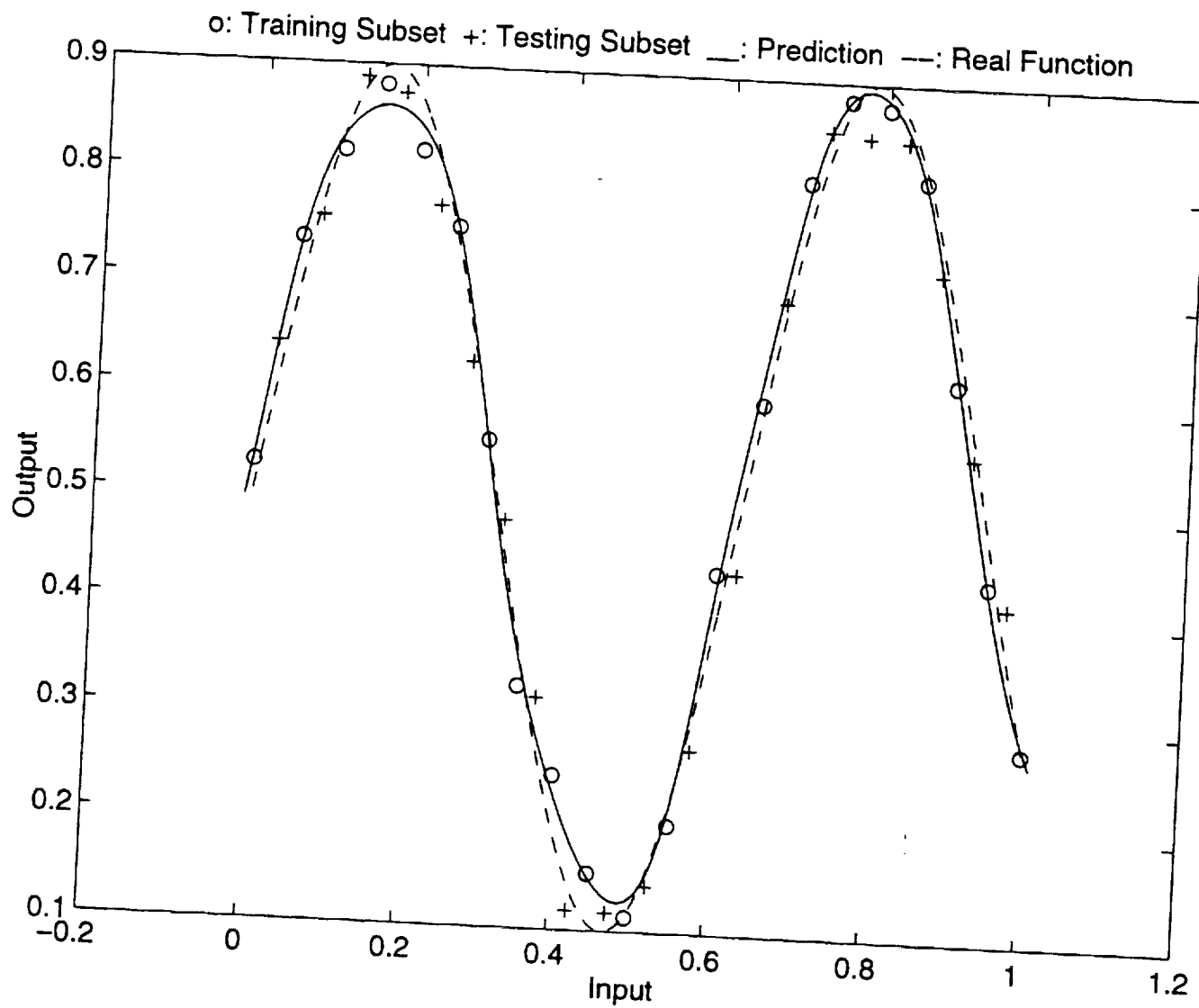


Fig. 26(a) Training History for the Toy Problem by a (1-50-1) NN:
Combined Errors

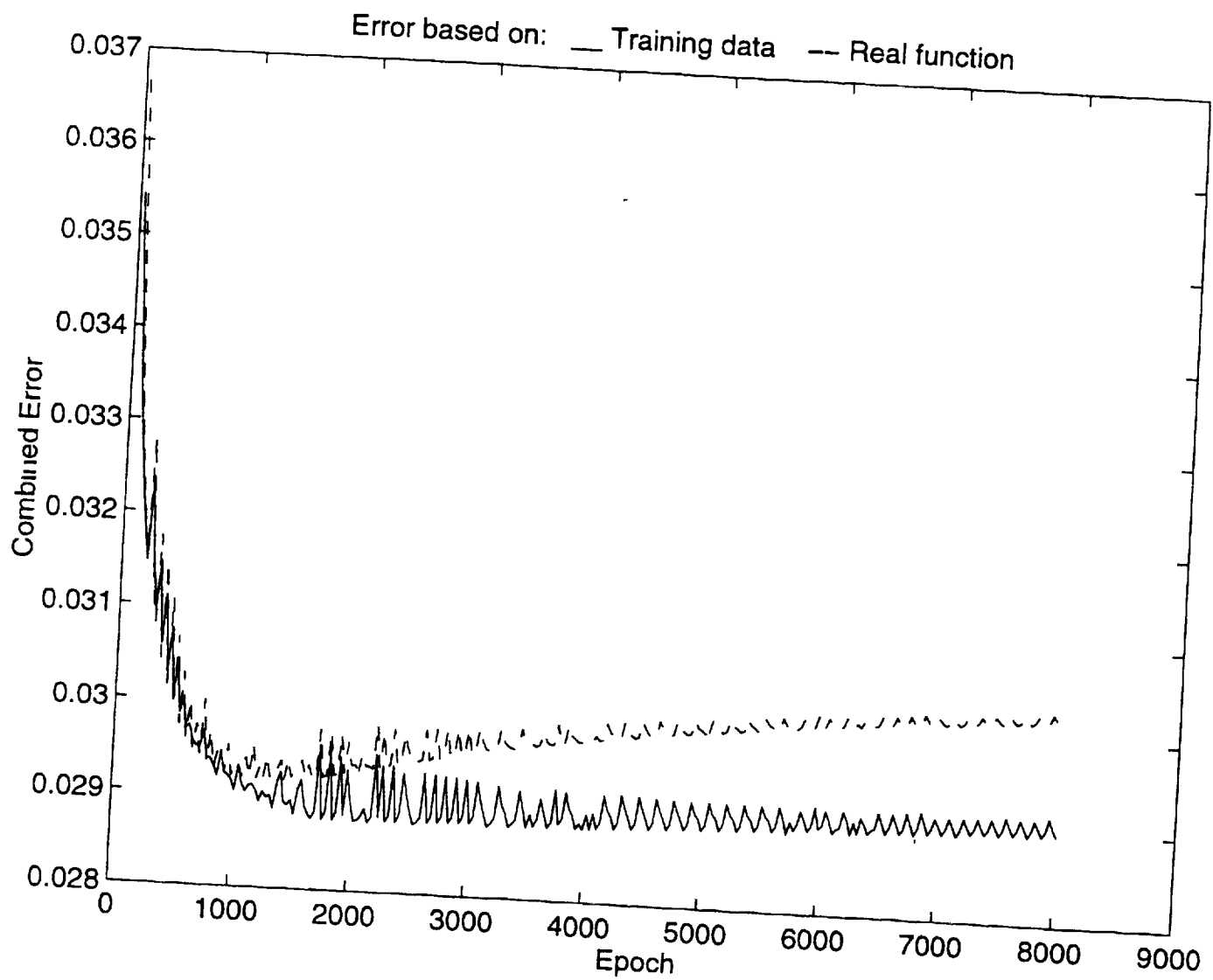


Fig. 26(b) Training History for the Toy Problem by a (1-50-1) NN:
Ratios of Testing Error over Training Error

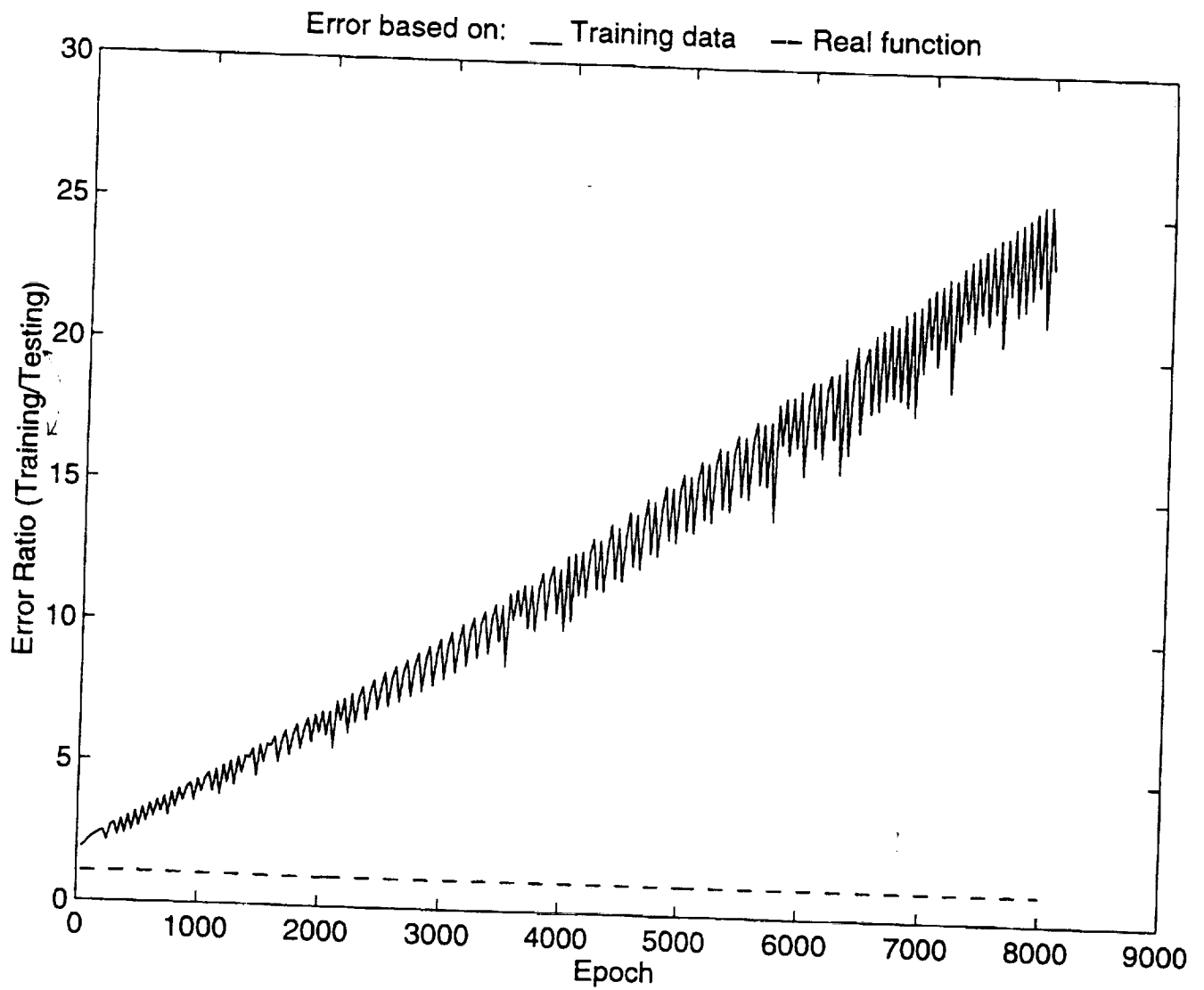


Fig. 26(c) Results for the Toy Problem by a (1-50-1) NN:
Comparison of the Prediction and the Target Values

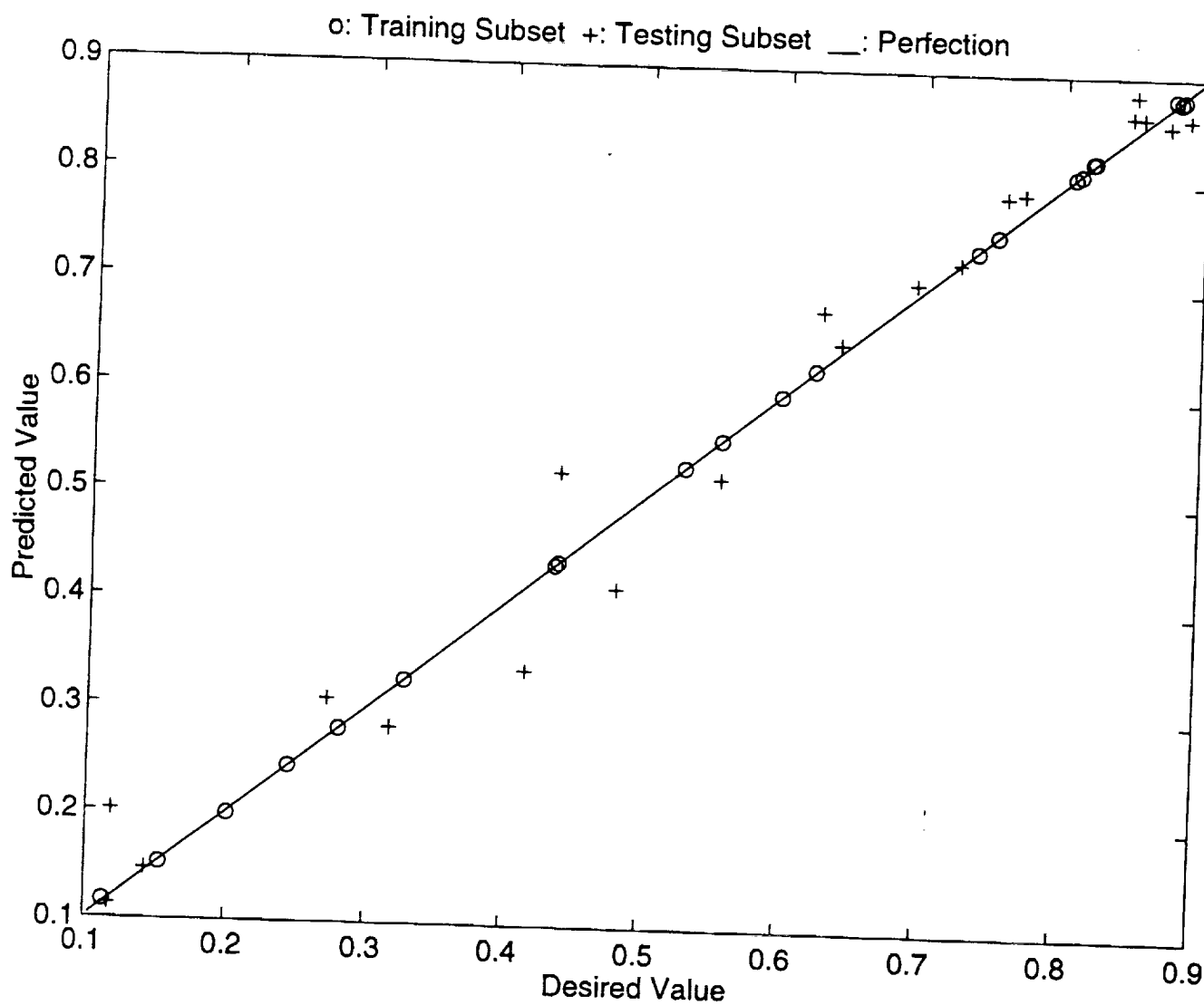


Fig. 26(d) Results for the Toy Problem by a (1-50-1) NN: Comparison of the Prediction, Training and Testing Subsets

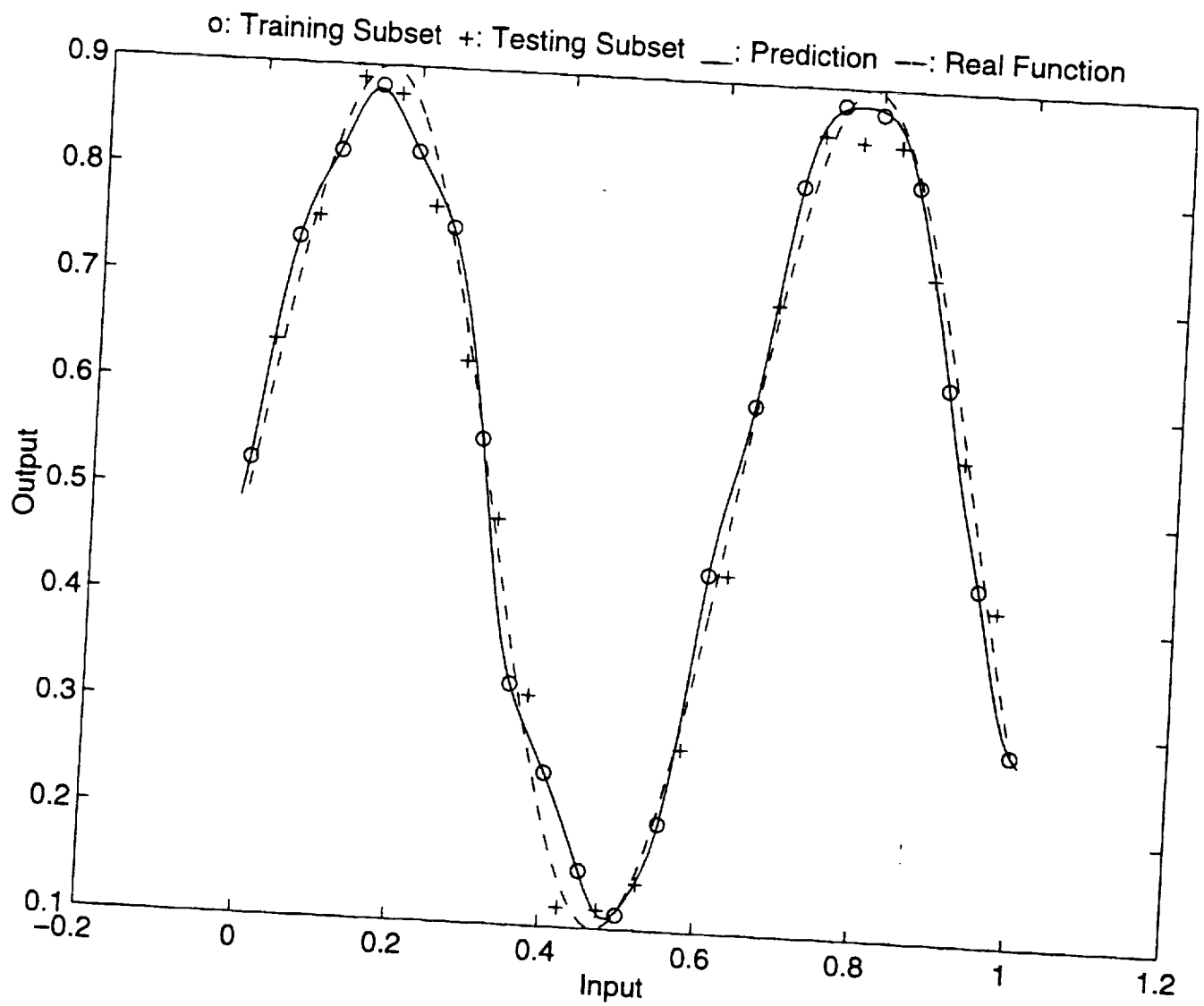


Fig. 27(a) Training History for the Toy Problem by a (1-100-1) NN:
Combined Errors

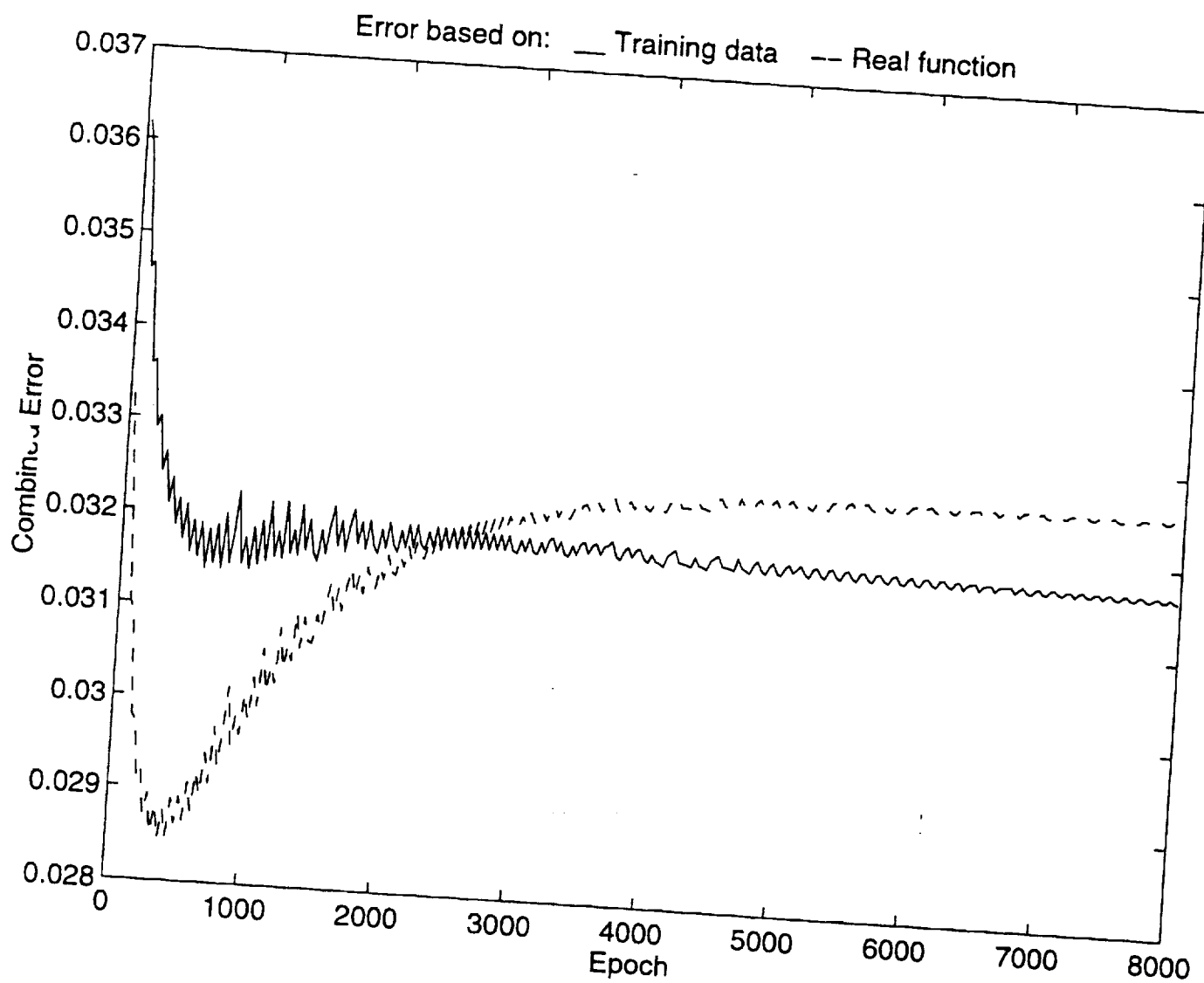


Fig. 27(b) Training History for the Toy Problem by a (1-100-1) NN:
Ratios of Testing Error over Training Error

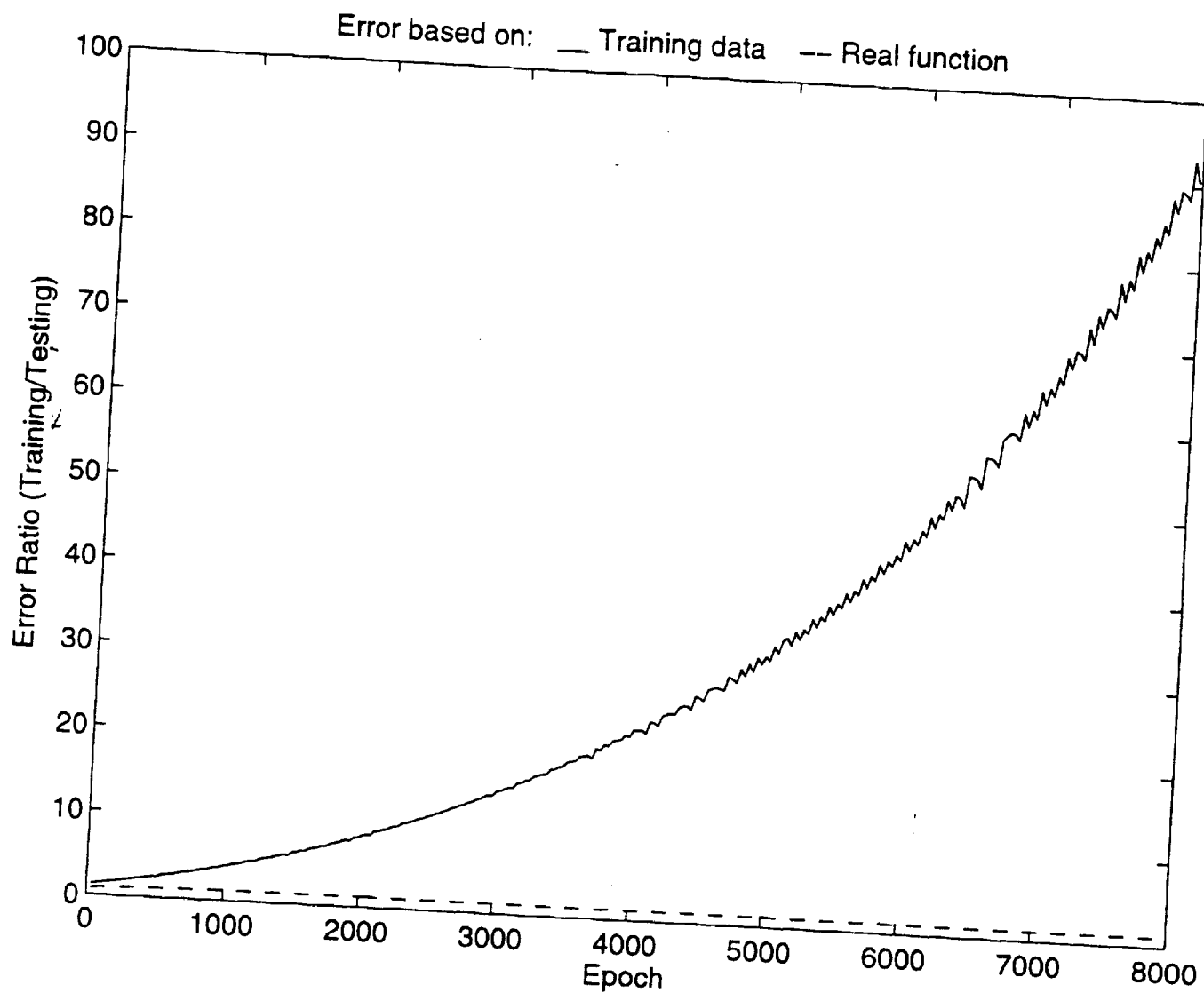


Fig. 27(c) Results for the Toy Problem by a (1-100-1) NN:
Comparison of the Prediction and the Target Values

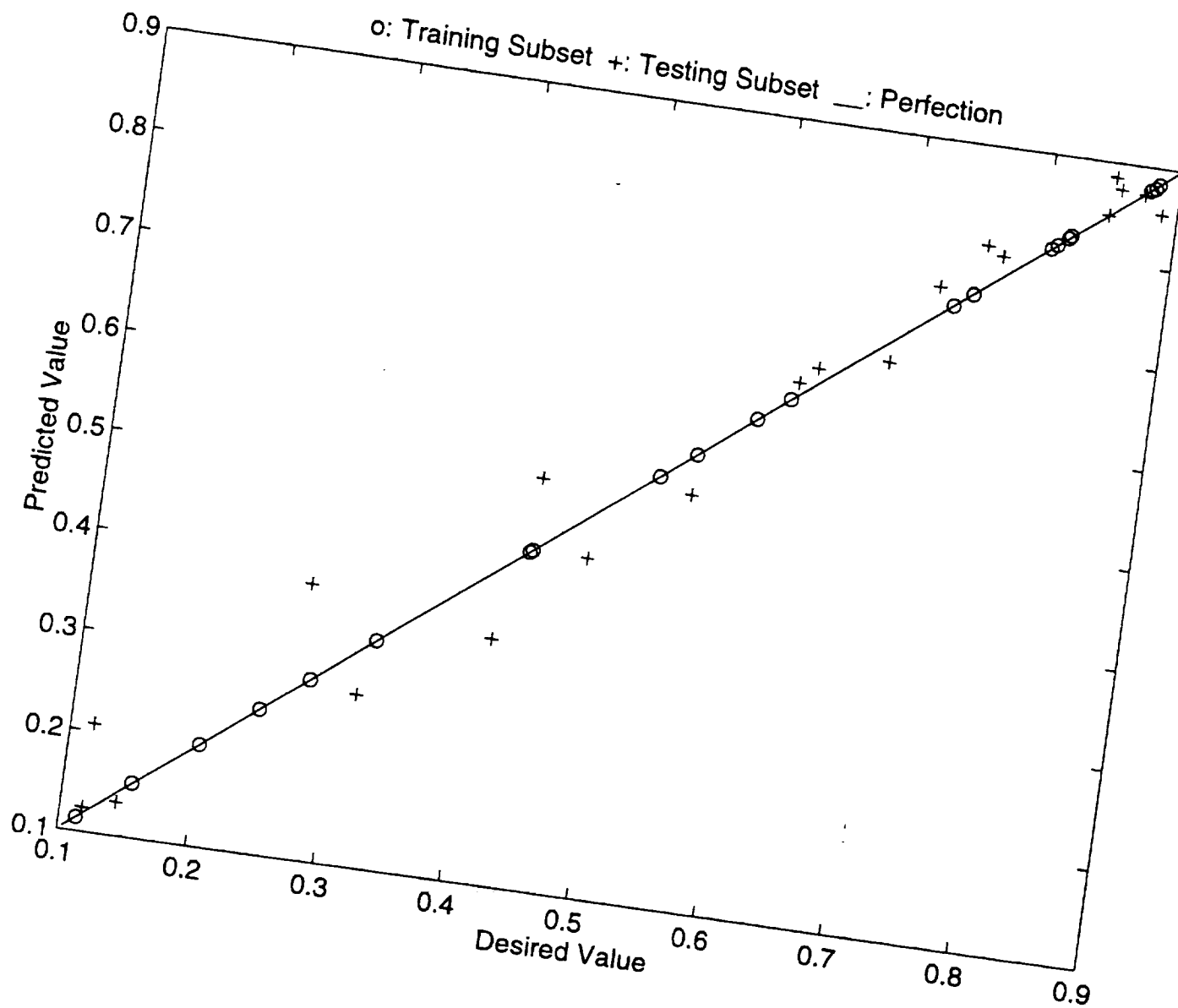


Fig. 27(d) Results for the Toy Problem by a (1-100-1) NN: Comparison of the Prediction, Training and Testing Subsets

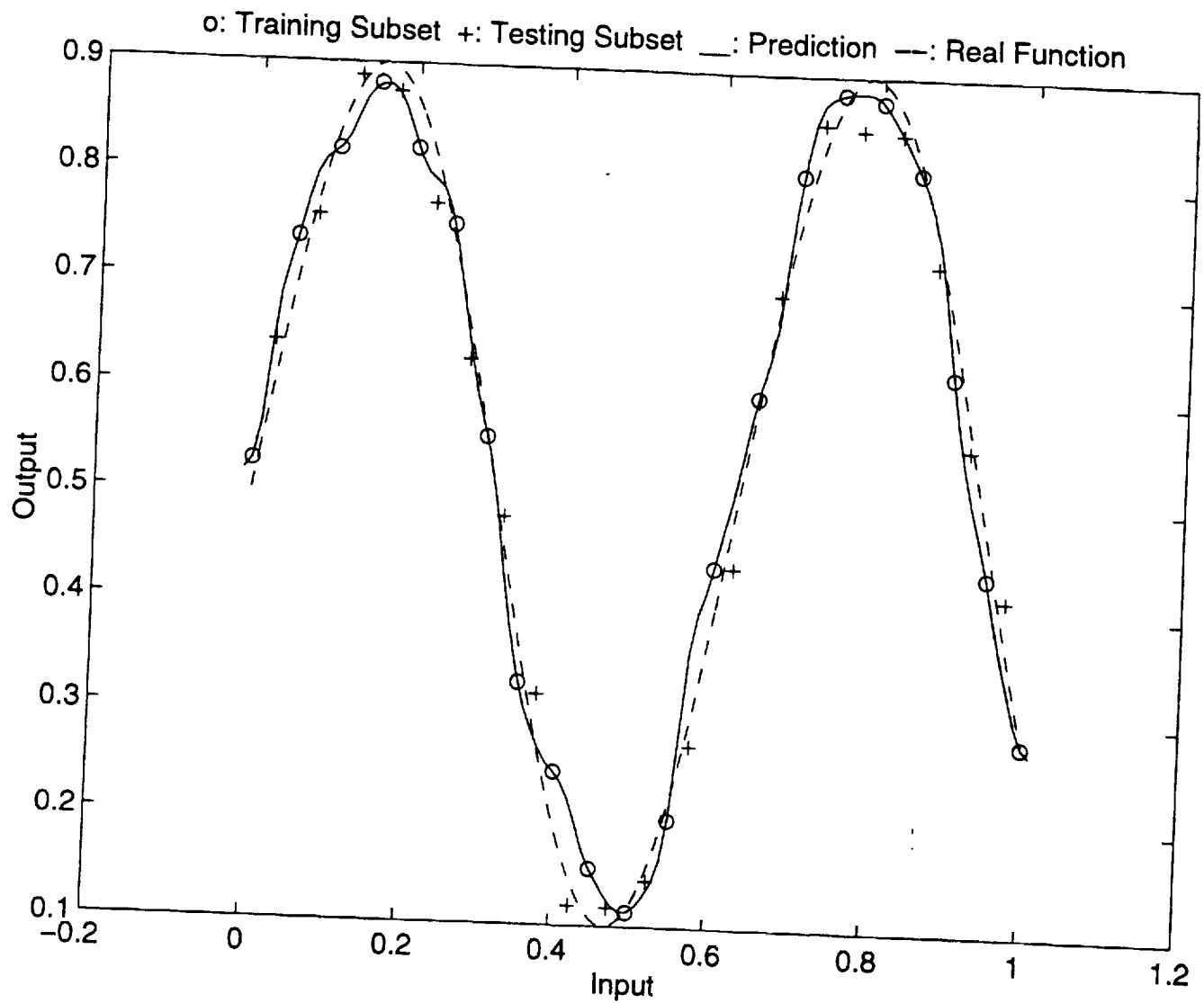


Fig. 28(a) Training History for the Toy Problem by a (1-20-20-1) NN:
Combined Errors

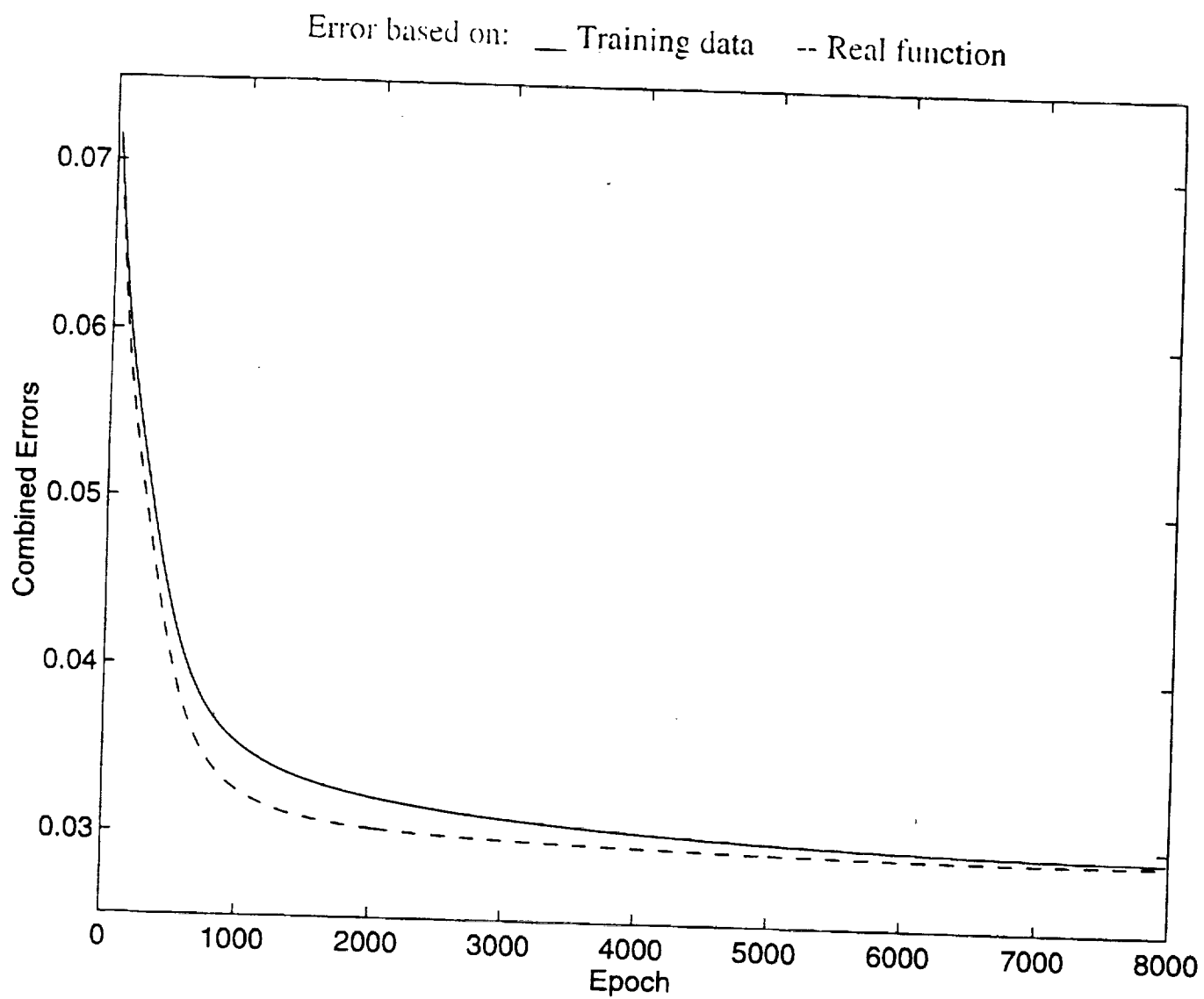


Fig. 28(b) Training History for the Toy Problem by a (1-20-20-1) NN:
Ratios of Testing Error over Training Error

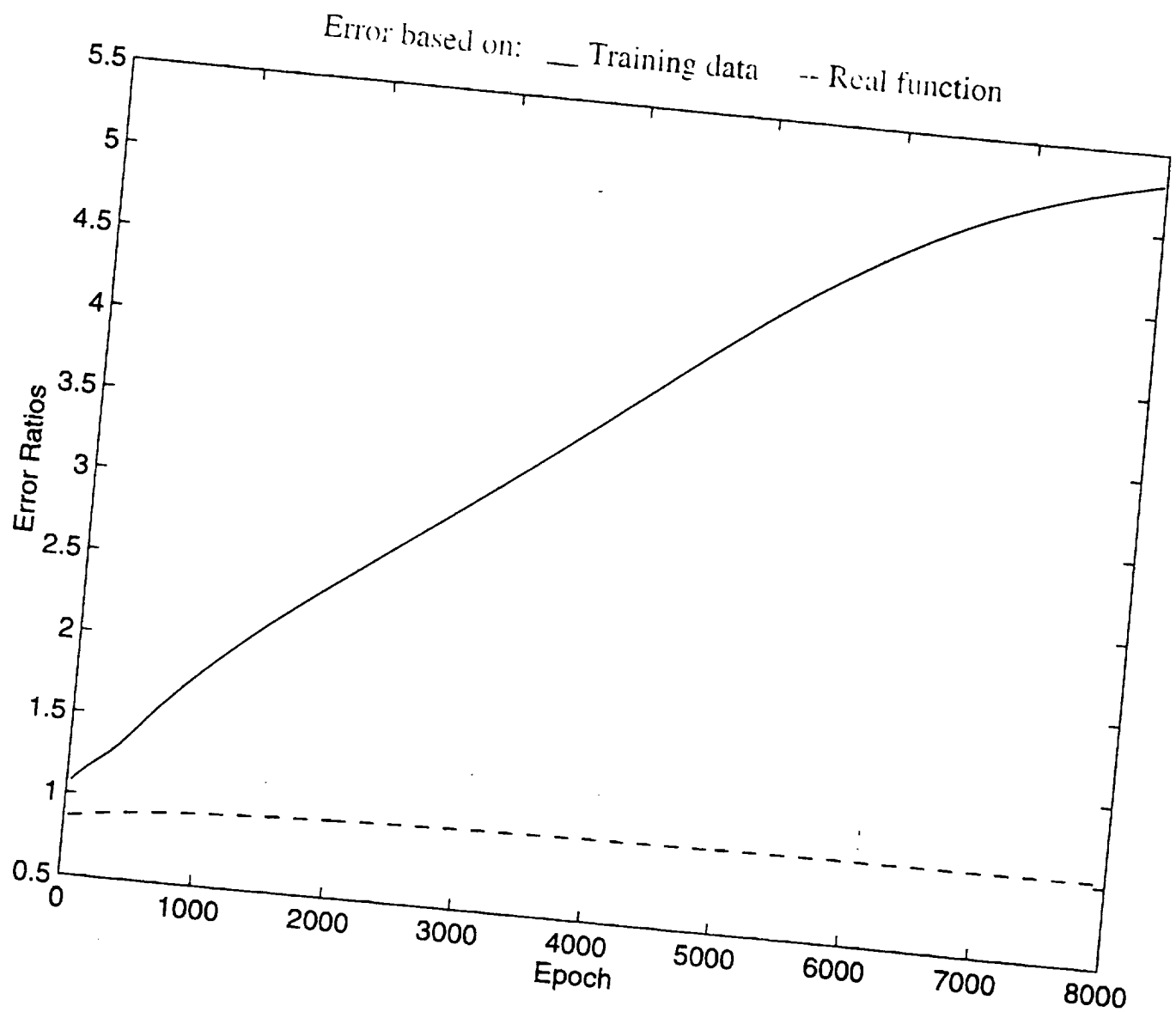


Fig. 28(c) Results for the Toy Problem by a (1-20-20-1) NN:
Comparison of the Prediction and the Target Values

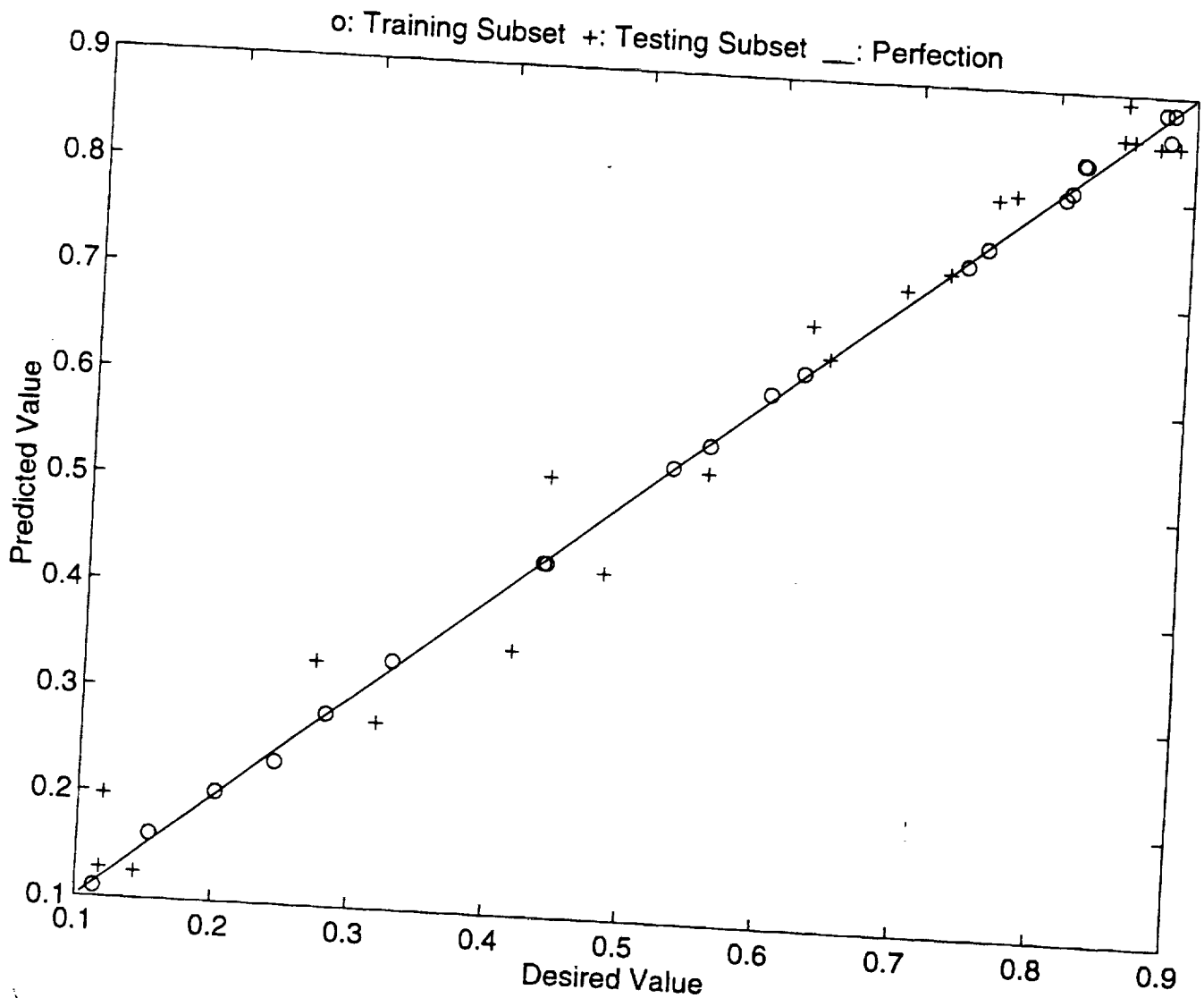


Fig. 28(d) Results for the Toy Problem by a (1-20-20-1) NN: Comparison of the Prediction, Training and Testing Subsets

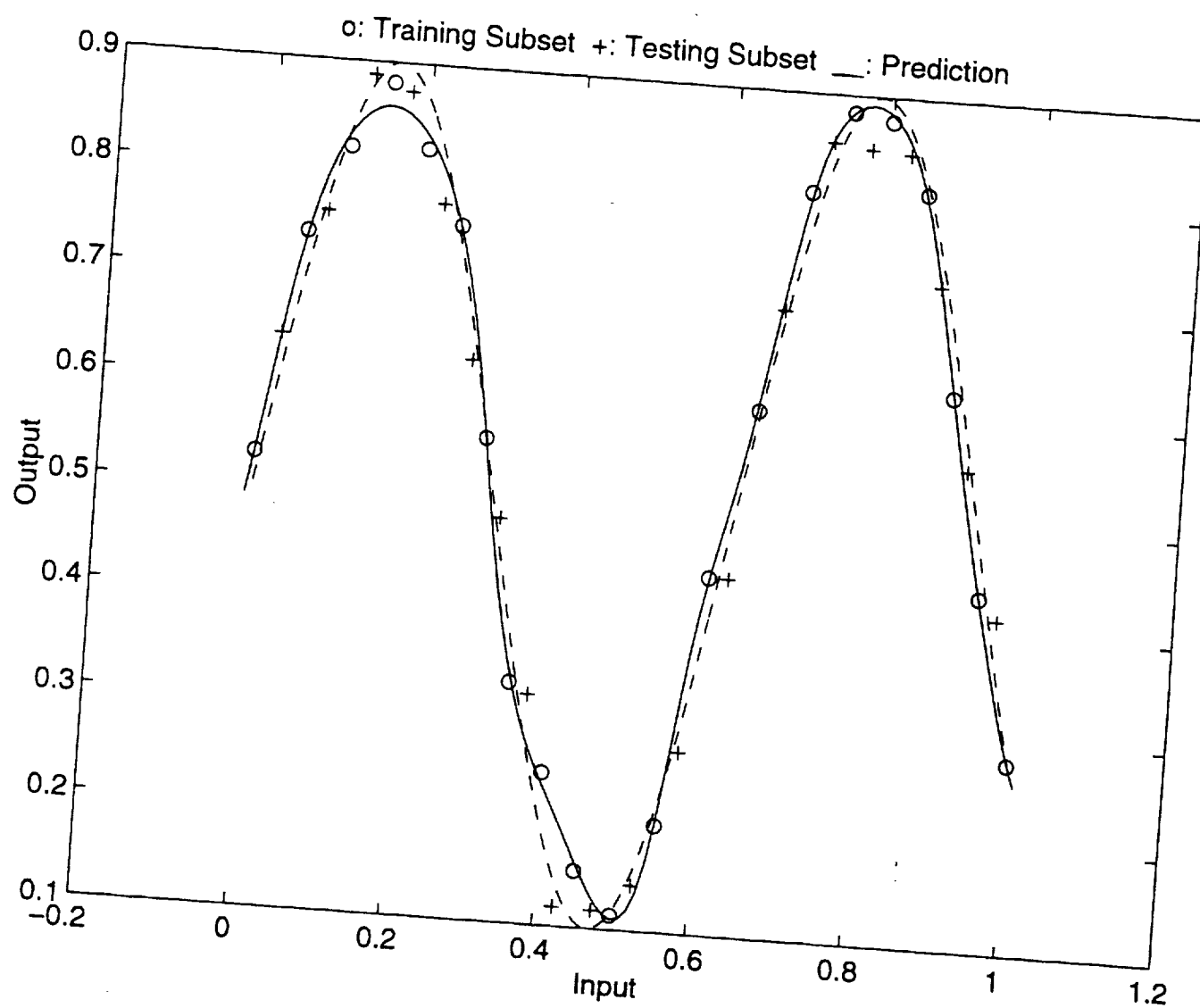


Fig. 29(a) Training History for the Toy Problem by a (1-50-50-1) NN:
Combined Errors

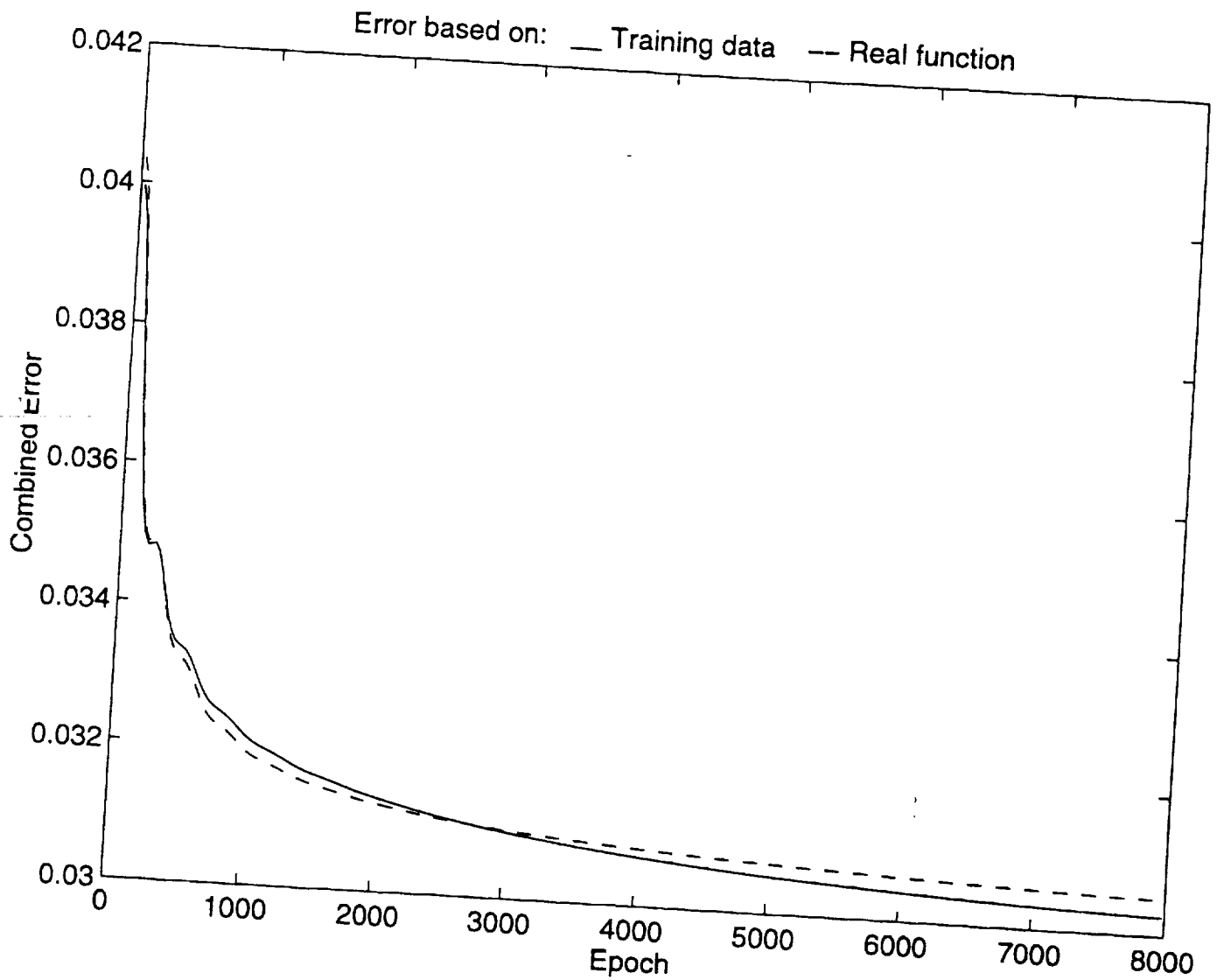


Fig. 29(b) Training History for the Toy Problem by a (1-50-50-1) NN:
Ratios of Testing Error over Training Error

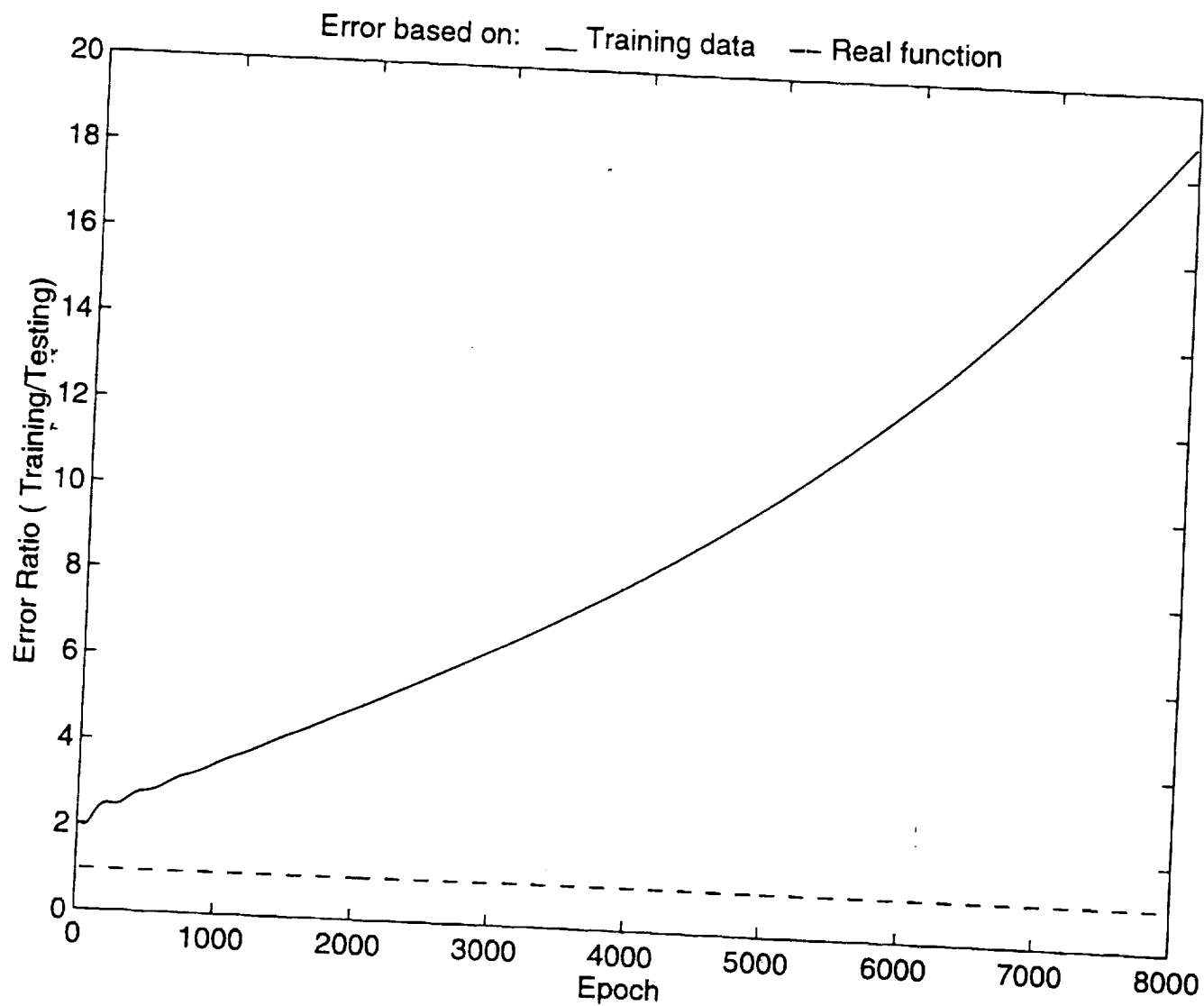


Fig. 29(c) Results for the Toy Problem by a (1-50-50-1) NN:
Comparison of the Prediction and the Target Values

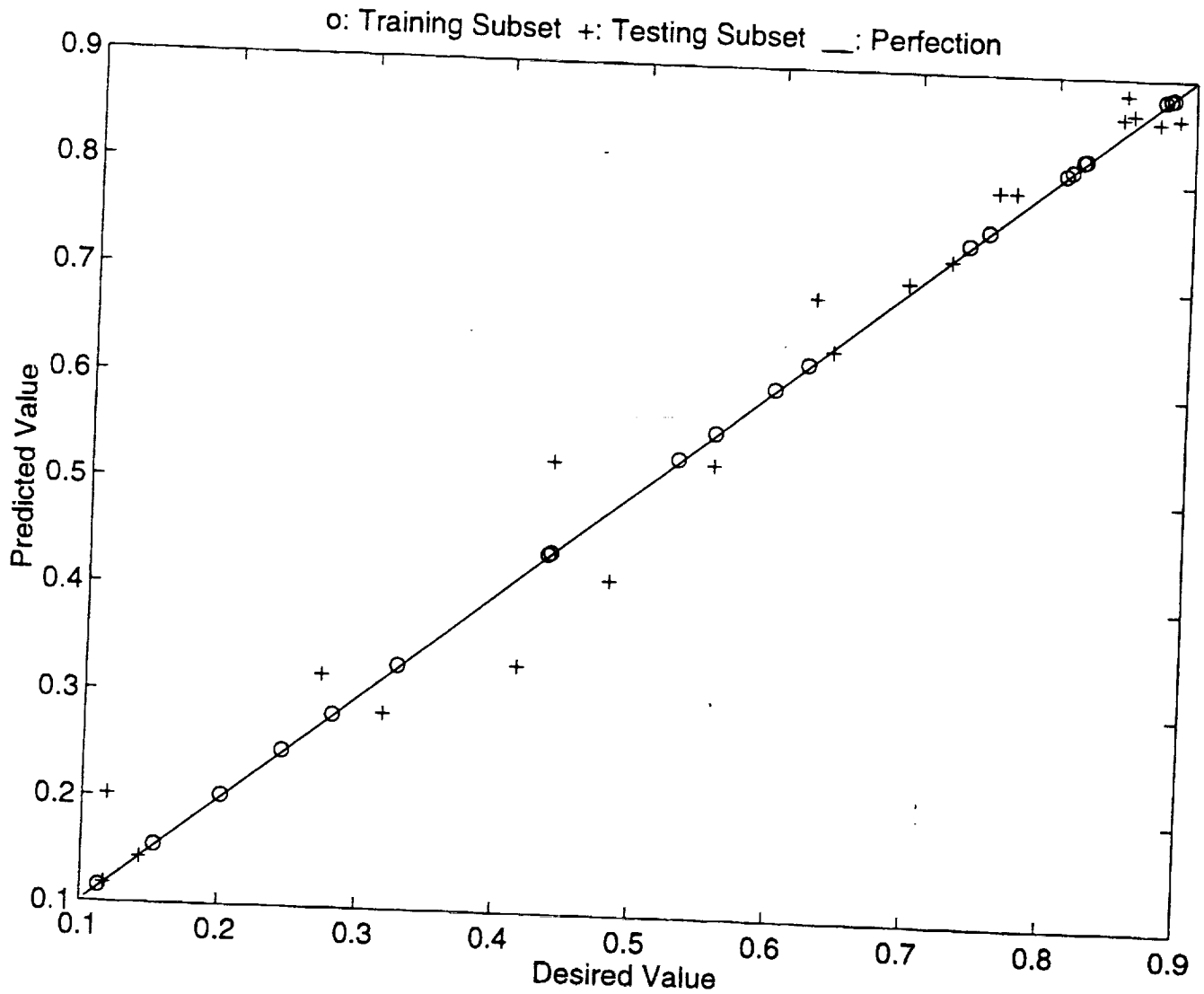


Fig. 29(d) Results for the Toy Problem by a (1-50-50-1) NN: Comparison of the Prediction, Training and Testing Subsets

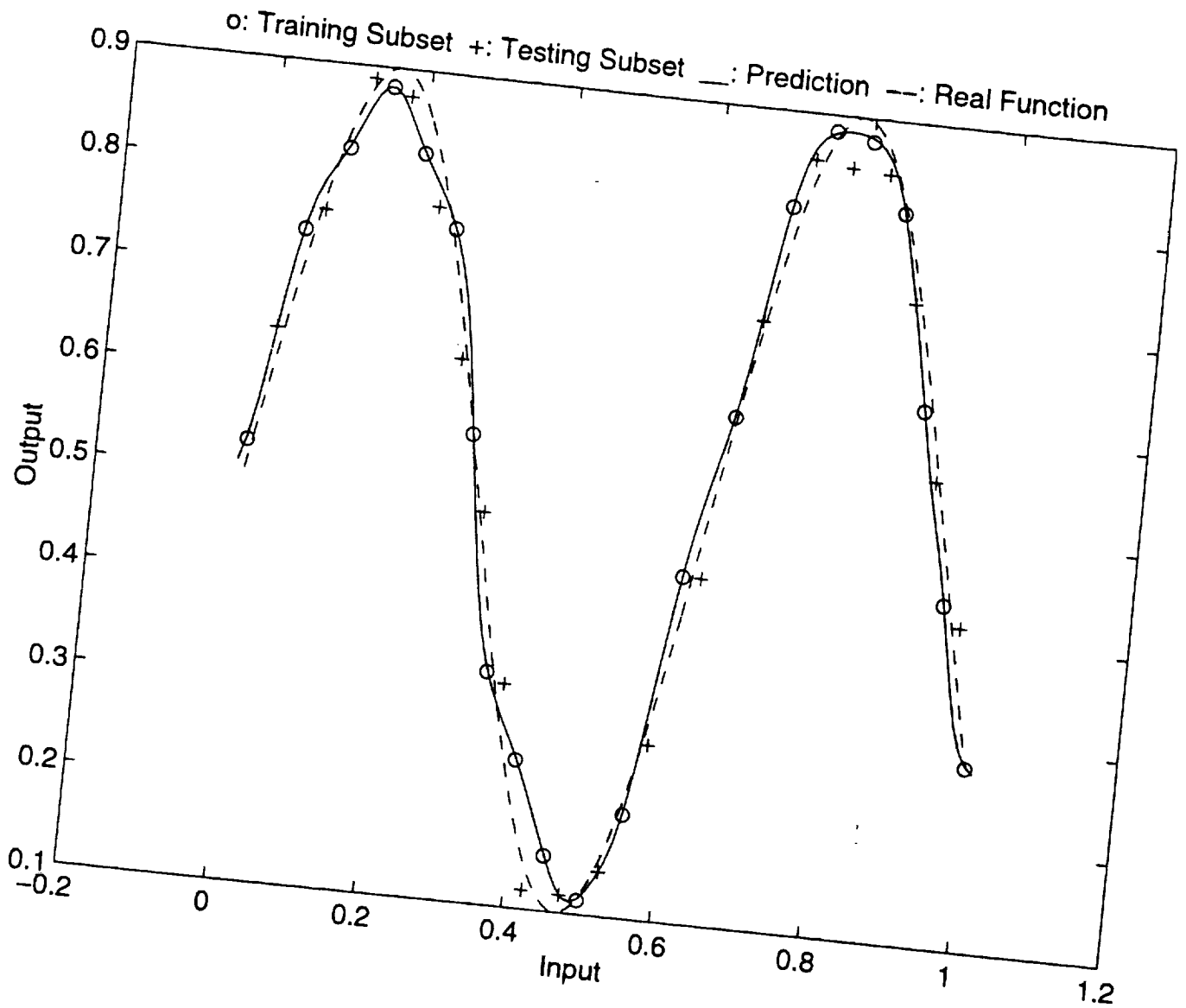


Fig. 30(a) Training History for Problem 5.2.1 by a (5-50-50-1) NN

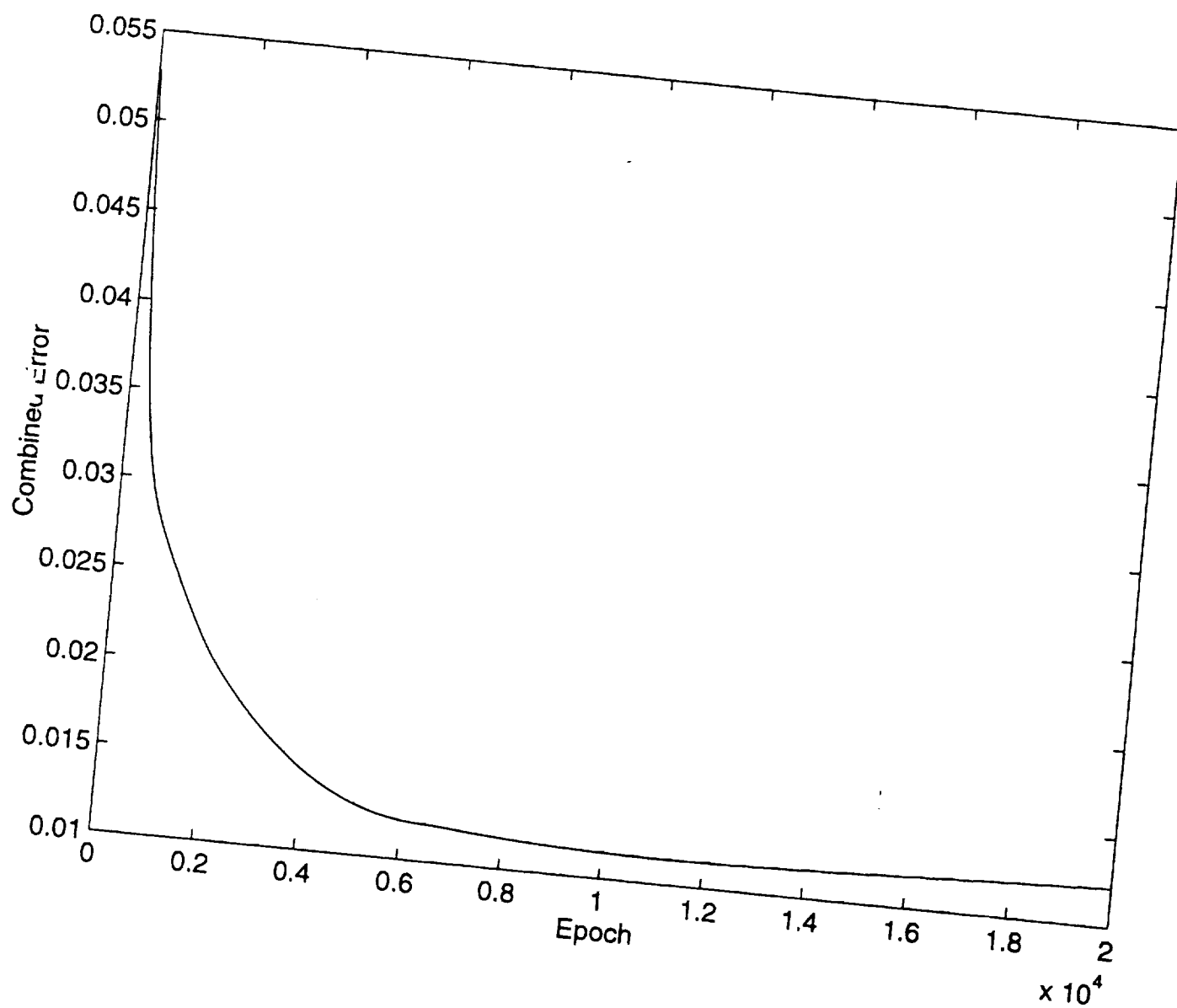
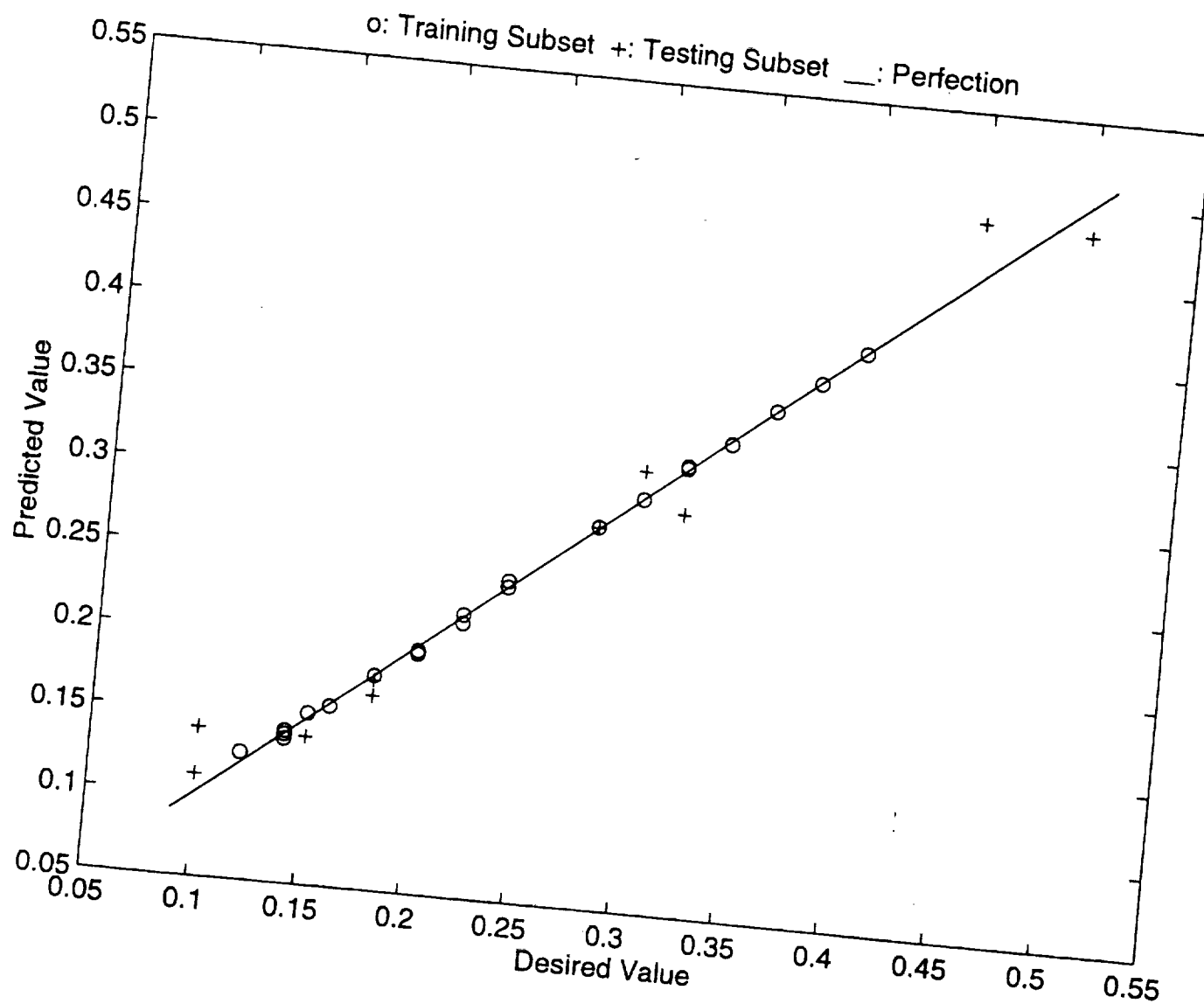
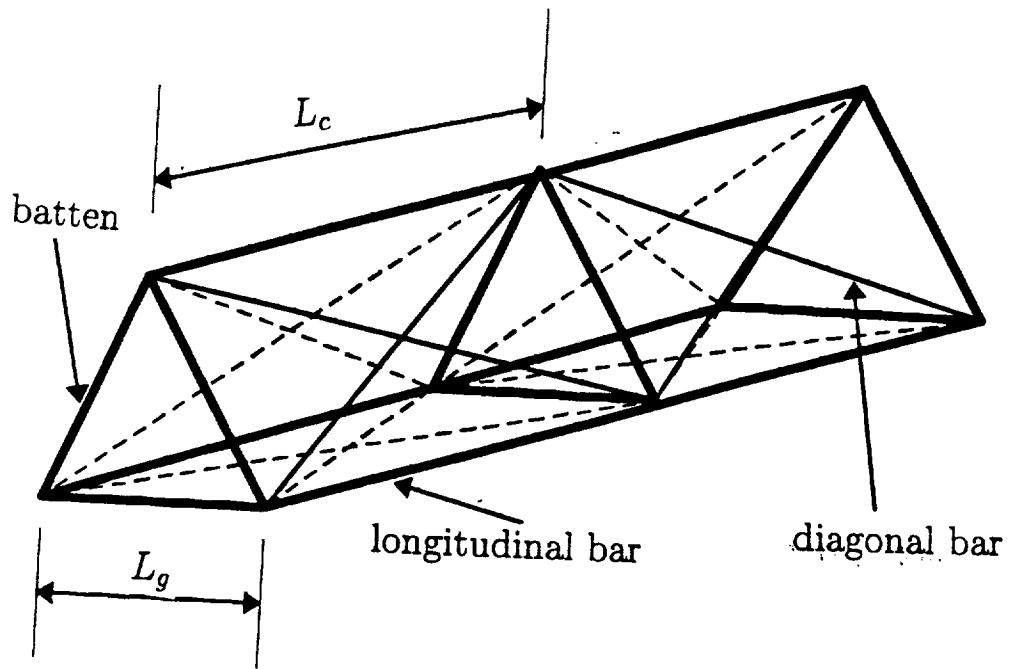


Fig. 30(b) Results for the Problem 54.2.1 by a (5-50-50-1) NN:
Comparison of the Prediction and the Target Values





Geometry of repeating cells of a
single-bay double-laced lattice
structure

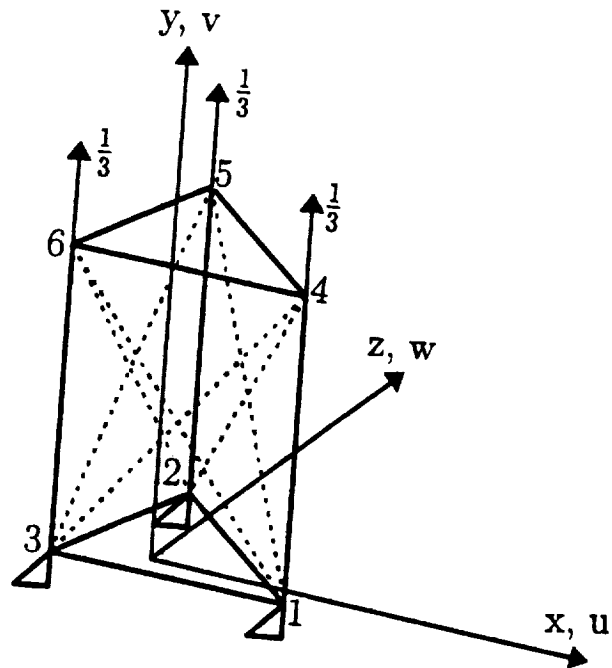
Length of longitudinal bars: L_c

Length of battens: L_g

Length of diagonal bars: $L_d = (L_c^2 + L_g^2)^{1/2}$

Areas: A_c, A_g, A_d .

Fig. 31

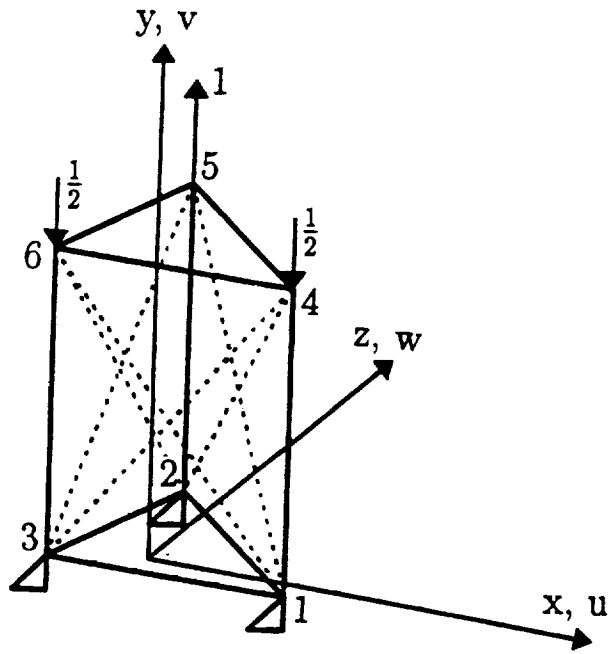


Evaluate Continuum Model Properties

Axial Rigidity EA :

$$EA = \frac{L_c}{\Delta v} \cong \frac{L_c}{\frac{1}{4}v_4 + \frac{1}{2}v_5 + \frac{1}{4}v_6}$$

Fig. 32 (a)



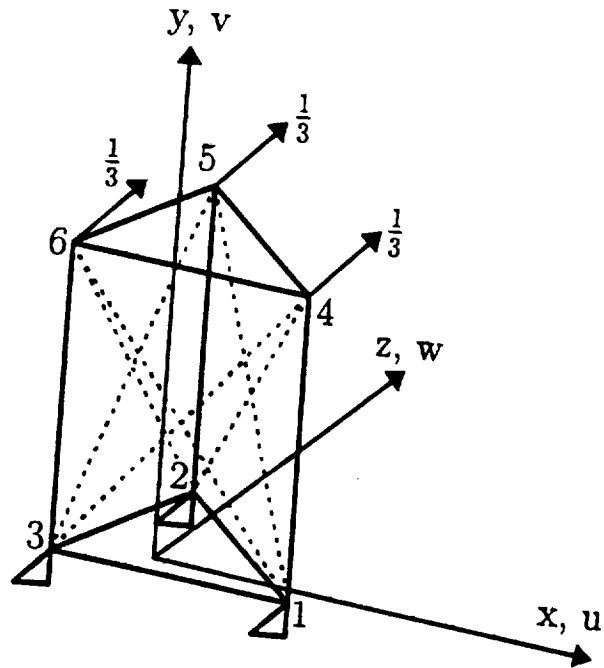
Evaluate Continuum Model Properties

Bending Rigidity EI :

$$M = 1 \times \frac{\sqrt{3}}{2} L_g = EI \cdot \frac{\phi}{L_c}$$

$$\Rightarrow EI = \frac{\frac{\sqrt{3}}{2} L_g L_c}{\phi} \cong \frac{\frac{3}{4} L_g^2 L_c}{v_5 - v_4}$$

Fig. 32 (b)



Evaluate Continuum Model Properties

Shearing Rigidity GA :

$$GA = \frac{1}{\theta} \cong \frac{1}{\frac{1}{4}\theta_4 + \frac{1}{2}\theta_5 + \frac{1}{4}\theta_6} = \frac{L_c}{\frac{1}{4}w_4 + \frac{1}{2}w_5 + \frac{1}{4}w_6}$$

Fig. 32 (c)

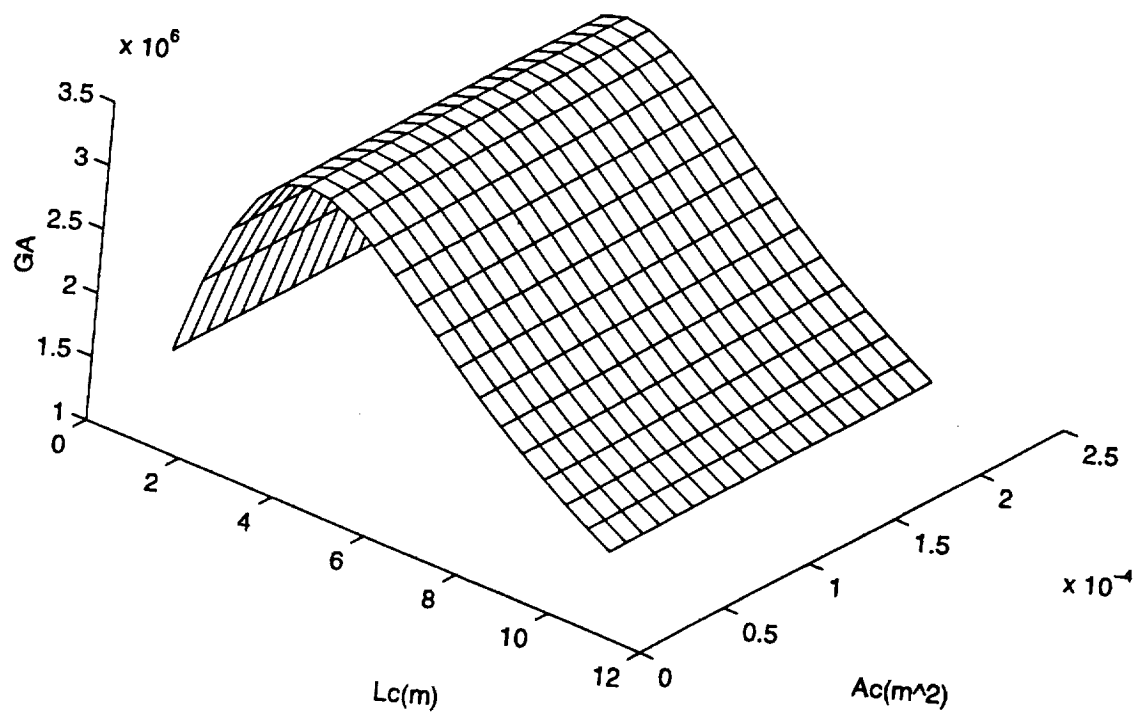


Fig. 33 $GA = f(A_c, L_c)$: (a) Training data

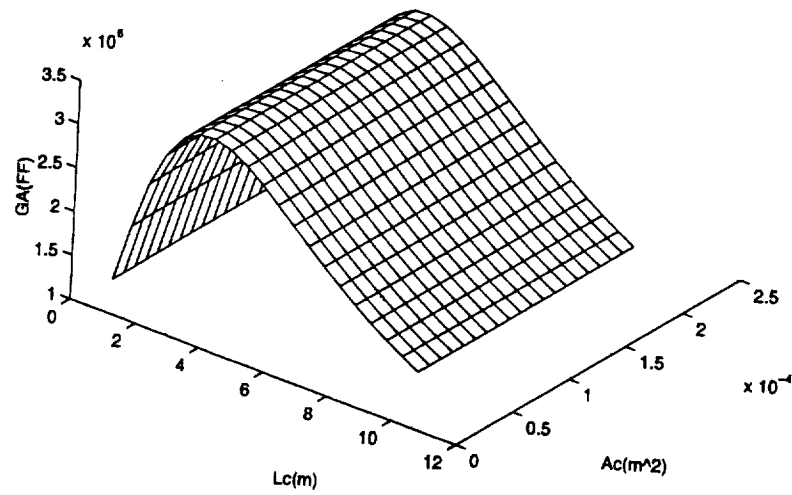


Fig. 33 $GA = f(A_c, L_c)$: (b) FF simulation on testing data

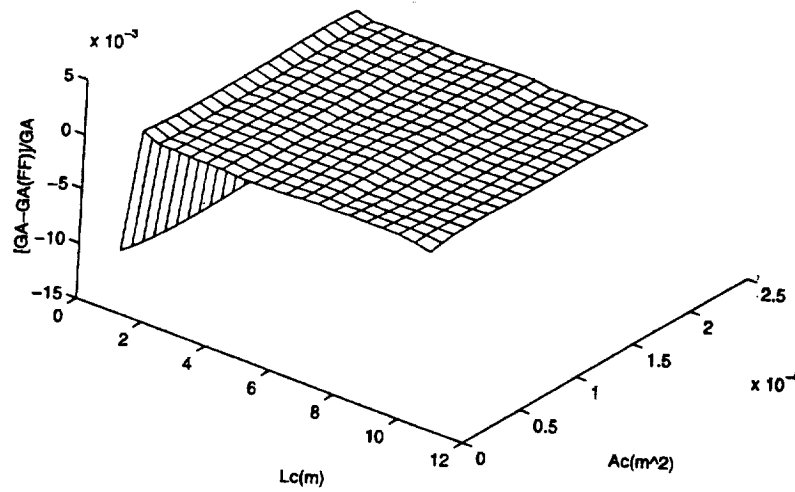


Fig. 33 $GA = f(A_c, L_c)$: (c) FF simulation errors

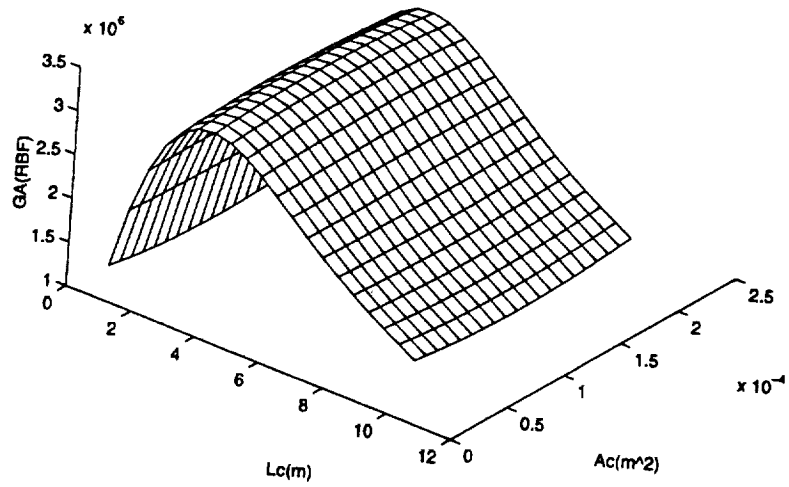


Fig. 33 $GA = f(A_c, L_c)$: (d) RBF simulation on testing data

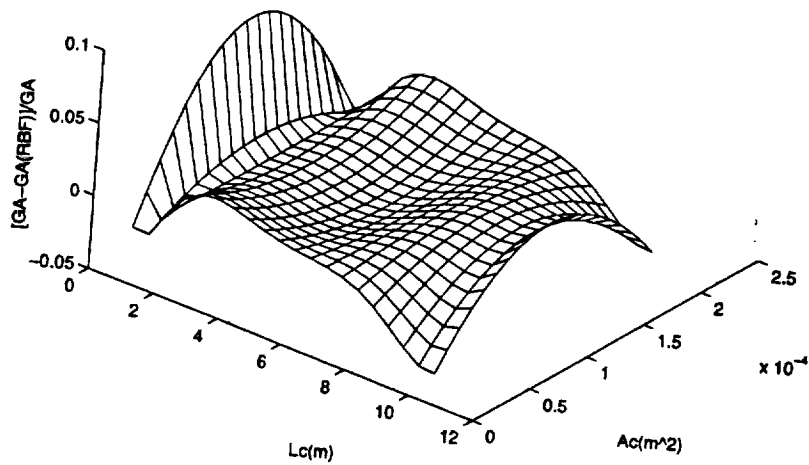


Fig. 33 $GA = f(A_c, L_c)$: (e) RBF simulation errors

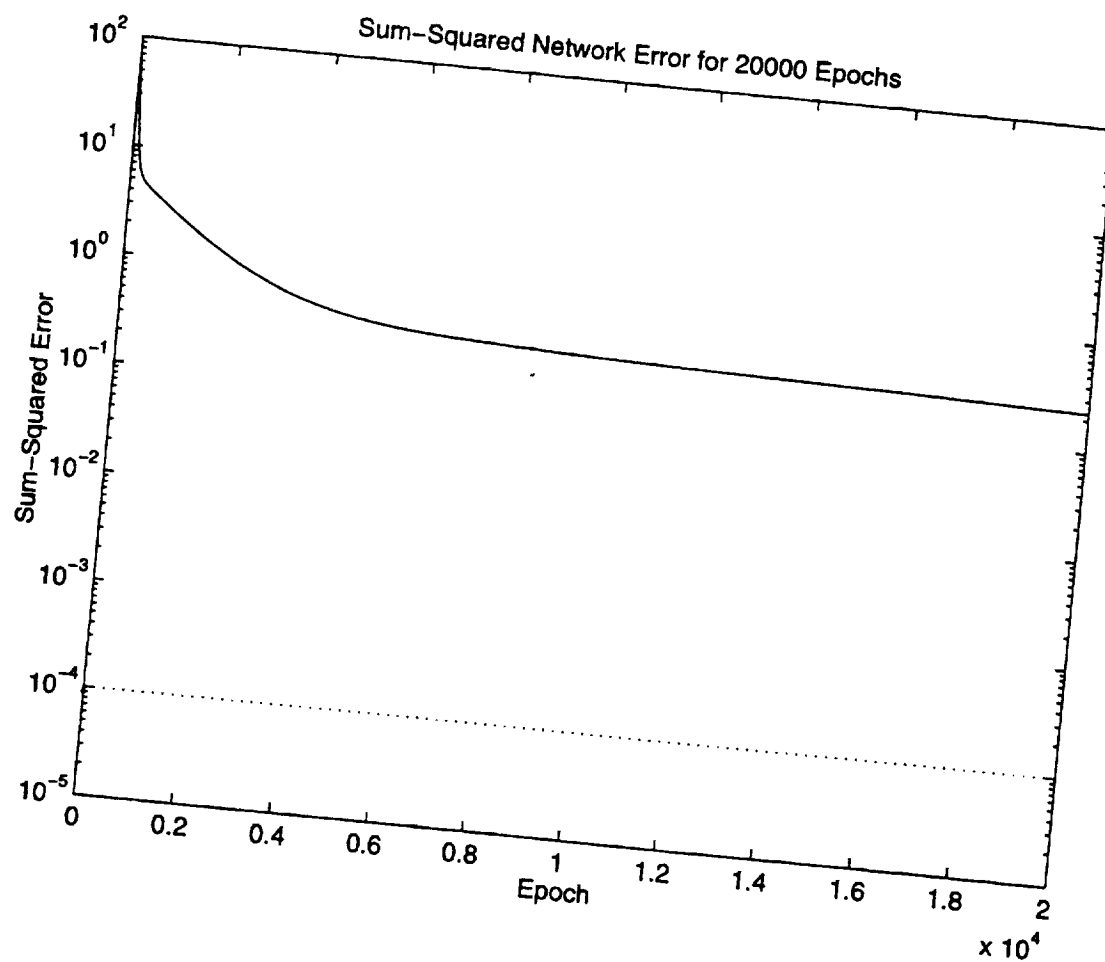


Fig. 34 (a) Training history of a 3-10-1 FF NN by *trainbp*

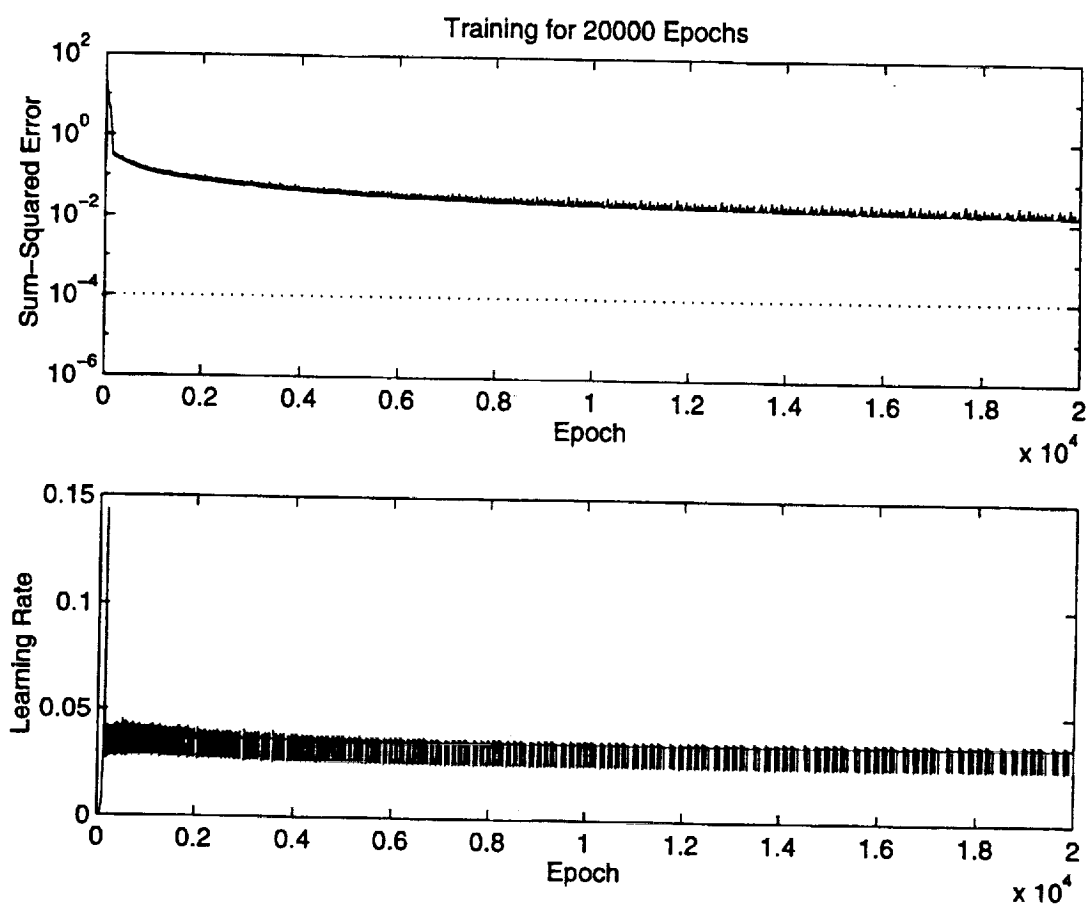


Fig. 34 (b) Training history of a 3-10-1 FF NN by *trainbpa*

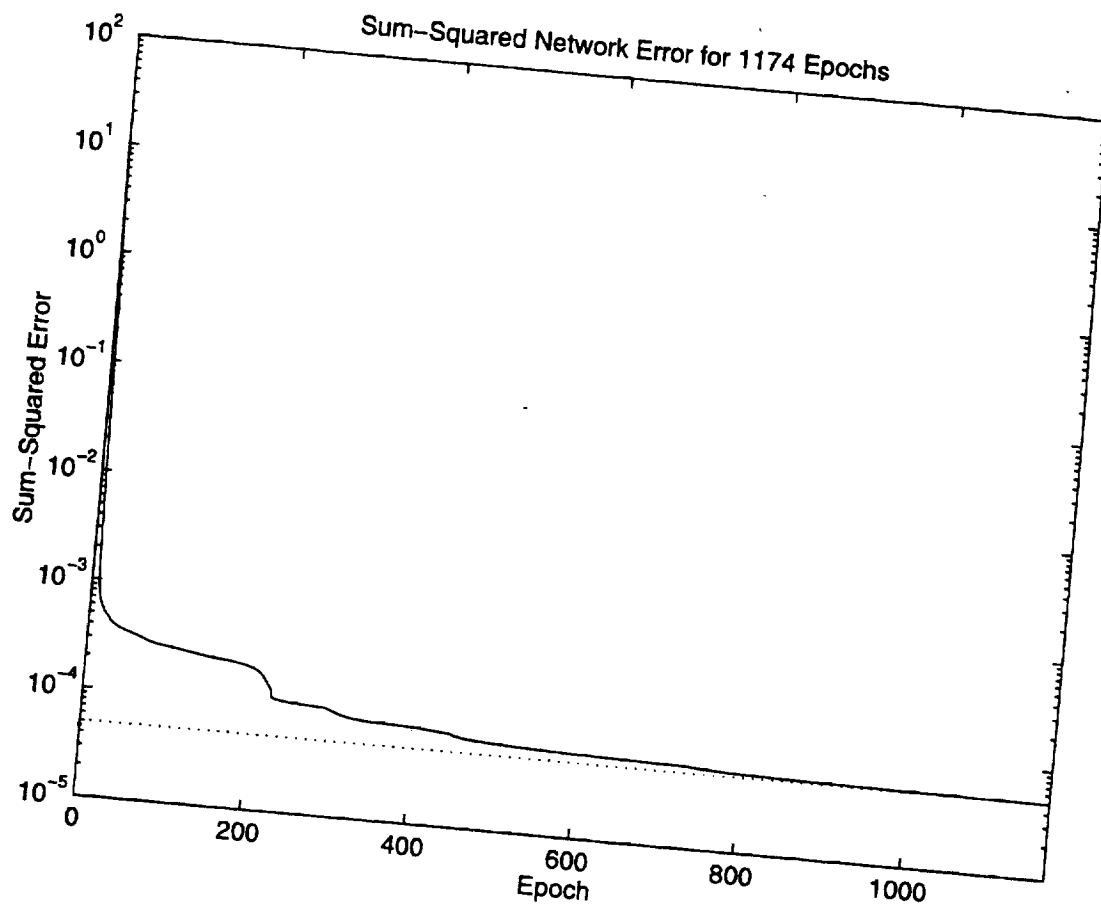


Fig. 34 (c) Training history of a 3-10-1 FF NN by *trainblm*